# SystemVerilog: Reusable Class Features, and Safe Initialization of Static Variables

Will Adams

Advanced Micro Devices, Inc

September 29, 2016

SNUG Austin

# Reusable Class Features

# Agenda

Features and mixins

Reusable features for SystemVerilog

Example: common configuration field for UVM components

Combining features

Conclusions

# Features

- A *feature* is a set of one or more properties in a class, together with the methods used to access them.
- Examples
  - **Name**: a string to store the value of the name, plus methods to get and (optionally) set the value.
  - **Id**: variable(s) to store the value, plus methods to access the value and parts of the value, and to format the ID for printing.
- Different classes have different sets of features.
  - For example, classes **A** and **B** have feature **Name**, classes **A** and **C** have feature **Id**.

# Sharing feature implementation

- A shared implementation of a feature ensures common behaviour, and makes it easy to modify the feature consistently.
- A *mixin* is a feature implementation that can be used by different classes.
  - Some languages provide mixins as a language feature, other languages use general-purpose mechanisms to implement mixins.
  - In C++, mixins are simple base classes, classes inherit mixins for required features.

```
class name_feature { … } ;
class id_feature { … } ;
```

```
class A : public name_feature , public id_feature { … } ;
class B : public name_feature { … } ;
class C : public id_feature { … } ;
```

# Reusable features in SystemVerilog

- SystemVerilog only supports single inheritance for implementation.
  – Interface classes allow multiple inheritance of interfaces, but not implementation.
- Difficult to use single inheritance to create base classes for features.
  – Classes for **Name**, **Id**, and both difficult to code without duplicating code.
  – Cannot be used with classes that extend a different class.

**Solution: Use inheritance and type parameters**

- Technique is outlined on following slides using example of a common configuration field for UVM components.
- Paper shows another example, a reset port for UVM components.

# Example: common configuration field

- Extend class `uvm_component` with configuration field `enable`.

```
class enable_component
  #( bit INIT = 1'b1 )
  extends uvm_component ;

  `uvm_component_param_utils_begin( enable_component )
    `uvm_field_int( enable , UVM_DEFAULT )
  `uvm_component_utils_end

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

  function bit is_enabled( ) ; return enable ; endfunction

  local bit enable = INIT ;

endclass
```

# Generalize `enable_component`

- Class `enable_component` cannot be used for classes derived from `uvm_scoreboard`, `uvm_driver #(REQ,RSP)`, etc.
- Add a type parameter to enable this.

```
class enable_component
  #( type T = uvm_component , bit INIT = 1'b1 )
  extends T ;
  // …
endclass
```

# Using `enable_component`

- Class `enable_component #(T,INIT)` extends any UVM component type.

```
typedef
  enable_component #( uvm_monitor )
  ip_monitor ;

class ip_scoreboard
  extends enable_component #( uvm_scoreboard ) ;
  // …
endclass

class ip_driver
  #( type REQ = uvm_sequence_item , type RSP = REQ , bit EN_INIT = 1'b0 )
  extends enable_component #( uvm_driver #( REQ , RSP ) , EN_INIT ) ;
  // …
endclass
```

# Combining features

- When a set of feature classes with common constructor arguments are defined, classes can be created with any subset of the features.

```
class feature0 #( type T = uvm_component ) ; … endclass
class feature1 #( type T = uvm_component ) ; … endclass
class feature2 #( type T = uvm_component ) ; … endclass
```

```
class feature012_component
  extends feature0 #( feature1 #( feature2 ) ) ;

class feature02_monitor
  extends feature0 #( feature2 #( uvm_monitor ) ) ;

class feature12_driver
  #( type REQ = uvm_sequence_item , type RSP = REQ )
  extends feature1 #( feature2 #( uvm_driver #( REQ , RSP ) ) ) ;
```

# Reusable class features: conclusions

- Parameterized inheritance allows sharing of feature implementations in SystemVerilog.
- Technique is not a general as mixins, since the constructor arguments must match the base class.
  - Not an issue for UVM components, since these are required to have the same constructor arguments in order to support factory creation.
- When a set of feature classes with the same constructor arguments is defined, any subset can be included in a class.

# Safe Initialization of Static Variables

# Agenda

Problem with static variable initialization in SystemVerilog

Avoiding the problem

Solving the problem

Conclusions

# The static initialization problem

- When a SystemVerilog static variable is created, it is set to the default value for its type.
- Initialization (with a variable declaration assignment) is completed before any initial blocks are executed.
- Variables are initialized in arbitrary order.
- A static variable may be accessed before it has been initialized.

```
static bit T = 1'b1 ;
```

```
static string S = ( T ? "ENABLED" : "DISABLED" ) ;
```

- Values of `T` and `S` after initialization depend on initialization order.
  - `T` initialized first → `T = 1'b1` `S = "ENABLED"`
  - `S` initialized first → `T = 1'b1` `S = "DISABLED"`

# Avoiding initialization problems

- A static variable without a variable declaration assignment is set to the default value for its type when it is created, and is not further initialized.
- Variable is initialized when it is created, so no initialization issues.
- If possible, transform static variables to use default initialization.

```
static bit T = 1'b1 ;
```

→

```
static bit not_T ;
```

- Avoid initializing static variables with initial blocks.
  - Initial blocks are executed in arbitrary order.

# Static objects

```
class C ;
  function new( int i ) ; … endfunction
  // …
endclass
```

```
static C J = new( 17 ) ;
```

```
function C J( ) ;
  // C++ style
  static C jj = new( 17 ) ;
  return jj ;
endfunction
```

**UNSAFE**

- Static variable `J` is null before it is initialized.
- Replace static variable `J` with function `J( )` that returns a handle.
  - Replace all references to variable `J` with function call `J( )`.
- C++-style function is unsafe.
  - `J( )` returns null if it is called before `jj` has been initialized.

# Safe initialization of static objects

```
static C J = new( 17 ) ;
```

```
function C J( ) ;
  static C jj ;
  if ( jj == null )
  begin
    jj = new( 17 ) ;
  end
  return jj ;
endfunction
```

```
function static C J( ) ;
  if ( J == null )
  begin
    J = new( 17 ) ;
  end
endfunction
```

- Initialize variable in the function code, guarded by a first-time check.
- Store the static value in the implicit return variable, declared with static lifetime.

# Singleton class

```
class S ;

  static S Instance = new( ) ;

  function new( ) ;
    // Initialize object
  endfunction

  // …

endclass : S
```

```
class S ;

  static function static S Instance( ) ;
    if ( Instance == null )
    begin
      Instance = new( ) ;
    end
  endfunction

  local function new( ) ; … endfunction

  // …
endclass : S
```

- Replace static variable `Instance` with function `Instance( )`.
  - Function declaration requires `static` twice.
  - Singleton instance is `S::Instance( )`.
- Declare function `new` local, so instances cannot be created by other code.

# Static variables of built-in types

```
static int I = 6 ;
```

```
class IntValue ;

  int value ;

  function new( int init ) ;
    value = init ;
  endfunction

endclass

function static IntValue I( ) ;
  if ( I == null )
  begin
    I = new( 6 ) ;
  end
endfunction
```

- Define a class that contains a non-static variable, and create a static instance of the class.
- Static variable is `I( ).value`.

# Generic value class

```
class Value
  #( type T = int ) ;

  T value ;

  function new( T init ) ;
    value = init ;
  endfunction

endclass
```

- Class `Value #(T)` can be used for any built-in type.

```
function static Value #( int ) I( ) ;
  if ( I == null )
  begin
    I = new( 6 ) ;
  end
endfunction

function static Value #( real ) R( ) ;
  if ( R == null )
  begin
    R = new( 7.28 ) ;
  end
endfunction
```

# Restrict interface to value

- Instead of a variable, use an object that provides the appropriate interface for the variable.

```
class Counter ;

  function new( int unsigned i ) ; value_ = i ; endfunction ;
  function int unsigned value( ) ; return value_ ; endfunction ;
  function void incr( ) ; ++value_ ; endfunction ;
  function void reset( ) ; value_ = 0 ; endfunction ;

  local int unsigned value_ ;

endclass
```

# Safe initialization of static variables: conclusions

- Avoid using variable declaration assignments or initial blocks to initialize static variables.
- Use default initial value where possible.
- For a static object that requires initialization, define a function to initialize the object and return a handle to it.
- For a static variable of built-in type that requires initialization, use a static object.

# Thank You