Essential SystemVerilog for UVM

Lecture Manual

Table of Contents

Essential SystemVerilog for UVM

Essential SystemVerilog for UVM

About This Course

Module 1

Course Prerequisites

Before taking this course, you need to:

- Have some knowledge of SystemVerilog for verification

The following knowledge is useful, but not essential

- An appreciation of object-oriented design
- Some familiarity with the Cadence® simulator

Course Objectives

The focus of this course is to review SystemVerilog constructs and Object-Oriented Programming (OOP) for UVM

In this course, you

- Review SystemVerilog classes, including static members, aggregation, randomization and constraints.
- Review class inheritance, including polymorphism, casting and virtual methods.
- Explore key Object-Oriented techniques such as reference, shallow and deep operations, instance names and parent pointers.
- Create class-based verification components using a standard architecture.

Course Agenda

- About This Course
- Basic Classes and Randomization
- Overview of DUT
  - Simple Data Class Declaration
  - Simple Randomization and Constraints
- Static Properties and Methods
  - Static Properties and Methods (Optional)
- Inheritance and Polymorphism
  - Inheritance and Polymorphism
- Aggregate Classes
- Components
  - Building a Component Hierarchy
- Connection to a DUT
- Building a Verification Component
  - Completing the Verification Component

Basic Classes and Randomization

Module **2**

Revision 3.0
Version 1.2.5

Module Objectives

In this module, you will

- Use basic classes and class-based features of SystemVerilog including:
  - Simple classes
  - Dynamic properties
  - Simple class randomization
  - Randomization constraints

Simple Class and Terminology

```
class basicframe;

  bit[3:0] addr;
  bit[7:0] data;

  function void print();
    $display("addr:%h data:%h",addr,data);
  endfunction

endclass
```

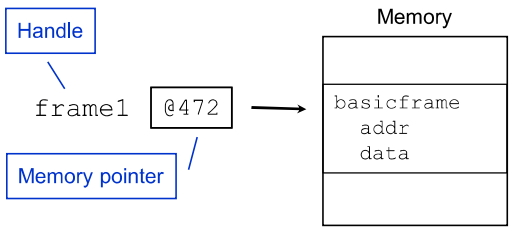
```
basicframe frame1;
basicframe frame2 = new();

initial begin
  frame1= new();
  frame1.addr = 5;
  frame1.data = 8'hf;
  frame1.print();
  ...
```

Declaration constructor

Statement constructor

addr:5 data:0f

- Class declaration
  - Properties (data)
  - Methods (subprograms)
- Full access to all members by default
- Object handle is a variable of the class type
  - Default value `null`
- Class instance created with constructor
  - Function `new`
  - Called inline with handle declaration...
  - ... or as procedural statement



8

The class basicframe contains class members addr, data, new and print. addr and data are class properties (data items). new and print are class methods (subprograms which operate on the class properties).

The class name is used as a type for declaring a class variable (handle). This variable defaults to null. A class instance is created by calling the class constructor, the method new. The constructor creates an instance of the class in memory and assigns the object handle a pointer to this memory location. A constructor is declared by default for all classes, or they can be explicitly declared as a method called new. An explicit class constructor follows all the normal rules for a SystemVerilog function, except it does not have a return type. Explicit constructors can be used to initialize class properties or indeed perform any class initialization requirement, including calling other class methods. Without an explicit constructor, class properties default to values based on their types (e.g., 'bx for logic properties, 'b0 for bit properties).

Instance Names

```
class namedframe;
  local string name;
        bit[3:0] addr;
        bit[7:0] data;

  function new(string name);
    this.name = name;
  endfunction

  function void print();
    $display("%s: addr:%h data:%h",name,addr,data);
  endfunction
...
```

string property

name argument to constructor

```
namedframe nframe;
namedframe farr [3:0];

initial begin
  nframe = new("nframe");
  nframe.name = "newname";
  foreach (farr [i]) begin
     farr[i] = new( $sformatf("farr[%0d]",i) );
     farr[i].print();
...
```

Handle array

Local property name not visible here

- Keep track of instances with string property
  - Instance name
  - Assigned in constructor
  - Added to `print()`
  - Declared as `local` to limit access
  - Additional house-keeping may be required
    - Methods to set/get name
- Can use string manipulation methods to generate names
  - e.g. `$sformatf`

```
farr[3]: addr:0 data:00
farr[2]: addr:0 data:00
farr[1]: addr:0 data:00
farr[0]: addr:0 data:00
```

9

We can name individual instances of a class to distinguish one instance from another and keep track of instances as they are used in a verification environment. We use a string property, assigned via an argument in an explicit constructor.

Remember the keyword this is a link to the current class instance. So this.name is a reference to the class property name which is otherwise hidden in new by the function argument name. Alternatively we could make the argument identifier different than the property name.

We include the name property in the print to help identify different instances.

Currently we can declare the name property as local (we will change this later). The local modifier means name is only visible in the class where it is declared, and not externally (e.g. via the handle nframe). This prevents the user from changing the name after it has been assigned. However if an instance is copied or cloned we may need to update the instance name. Therefore we may need to provide additional housekeeping methods to read or write the name. More on this later.

The instance name argument to the constructor is simply a string, so we can use string manipulation methods to, for example, assign different names to each element in an array of class instances. By using $sformatf, we can embed the array index in the instance name using a foreach statement.

Print Policy

```
typedef enum {HEX,BIN,DEC} pp_t;

class printframe;
  local string name;
        bit[3:0] addr;
        bit[7:0] data;

 function void print(input pp_t pp = HEX);
   $display("name: ",name);
   case (pp)
     HEX: $display("addr %h, data %h",addr,data);
     DEC: $display("addr %0d, data %0d",addr,data);
     BIN: $display("addr %b, data %b",addr,data);
   endcase
 endfunction
...
```

Policy values

Policy argument with default value

```
printframe pf;

initial begin
  pf = new("pf");
  pf.addr = 1;
  pf.data = 14;
  pf.print();
  pf.print(DEC);
...
```

```
name: pf
addr 1, data e
name: pf
addr 1, data 14
```

Need control over print format

- Use a control knob, or "policy"

Declare an `enum` type

- Outside class for visibility to instances
- Pass `enum` type as `print` argument
- Define a default value for usability

Policies are good for options which:

- Are small in number
- Unlikely to change

10

The print policy determines the print format. It is usually defined with an enumerate type, declared outside the class in order to give visibility to code which create class instances. A policy value of the enumerate type is passed into the print method as an argument. It may be convenient to define a default value for the policy argument, so that the print can be called without arguments.

Policies are good when there are a small, set number of options to choose from, which are unlikely to change. Therefore choosing the radix of a print method is a good application. Setting the address of a data packet between 0 and 15 is not a good policy application, as there is a large number of values, and the packet address bounds may change in future designs. Selecting packet address is better done with randomization.

Variable Data Length

```
class varframe;
 local string name;
       bit[3:0] addr;
 local bit[3:0] length;
       bit[7:0] payload[];

 function new(string name, bit[3:0] length);
   this.name = name;
   this.length = length;
   payload = new[length];
   foreach (payload[i])
     payload[i] = $urandom;
 endfunction

 function void print(...);
   $display("%s: addr:%h length:%h", name, addr, length);
   foreach (payload[i])
     $display($sformatf("payload[%0d]:%h",i,payload[i]));
...
```

Payload size

Dynamic array

Length argument

Create array

Randomize contents

Print policy omitted...

? How can we randomize frame length?

Question

- Data byte length can vary
  - Use dynamic array
- Array length is declared as a separate property
  - Assigned in constructor
- Length could be defined as `local`
  - Prevent access from outside class
  - Maintain dependency with payload size

```
varframe vframe;

initial begin
  vframe = new("vframe",8);
  vframe.addr = 4;
  vframe.length = 0;
  ...
```

Local property `length` not accessible here

11

In practical terms, a frame containing a single data byte is unrealistic. It is more likely that data would contain multiple or even a variable number of bytes.

For a variable number of bytes we can use a dynamic array. This is an array which does not exist until it is explicitly created during runtime using the array method new. The length of the array is set as an argument to the constructor.

So now we replace the single byte data property with a variable length payload property. We also add a new property length which will define the number of data bytes in the frame. payload is created in the class constructor and is set to the length of length, passed as a constructor argument. The constructor also initializes all payload bytes to random values using $urandom.

We also update the print method to print the data array contents using a foreach.

By default, all class properties are accessible outside the class declaration. However in varframe we have a dependency where length must be equal to the number of bytes in payload. Therefore, again, the modifier local is applied to the property. This prevents the user from writing directly to length from outside the class declaration. You could still provide write access to length via a method which would also update the size of the data array to maintain the dependency between the two properties.

Setting the payload length manually each time is tedious. Verification will be easier if we could create frames of random length.

Randomized Frame Length

```
class randframe;
       local string name;
       local bit[3:0] addr;
  rand local bit[3:0] length;
             bit[7:0] payload[];

 function new(string name);
   this.name = name;
 endfunction

 function void data_rand();
   payload = new[length];
   foreach (payload [i])
     payload[i] = $urandom;
 endfunction

  function void post_randomize();
    data_rand();
  endfunction
...
endclass
```

Randomize length

Create and randomize payload

Executed after successful randomization

- Create variable length frames by randomizing property **length**
  - No longer set in constructor
- Randomize affects **rand** properties only by default
  - Returns 1 on success
- **post_randomize** called after successful randomize
  - Here to create and load dynamic data array

```
randframe rframe;
int ok;

initial begin
  rframe = new("rframe");
  ok = rframe.randomize();
  ...
```

? Question

How can we avoid creating frames of length zero?

12

To test variable length frames, the length property value can be randomized. In the class randframe, the modifier rand is applied to the property length. The built-in class method randomize is called on instance rframe of class randframe, and this will assign a random value to length from its full range. If the randomization is successful, length is assigned and the randomize method returns the integer 1. Although the built-in randomize method cannot be overridden, we can declare post_randomize, which is automatically called after a successful randomize. Here post_randomize is used to create the payload dynamic array, and to randomize the array contents. These operations are performed in a separate method, data_rand, for flexibility (data_rand can be called separately to update the frame payload and can be separately overridden by a subclass).

However in the design specification, a frame must always contain at least one payload byte. As all values of length are equally likely, randomization could return 0. So we need to restrict or constrain the randomization of length to avoid 0.

In-Line Constraints

```
class randframe;
       local string name;
       local bit[3:0] addr;
  rand local bit[3:0] length;
             bit[7:0] payload[];
...
 function void data_rand();
   payload = new[length];
   foreach (payload [i])
     payload[i] = $urandom;
 endfunction

  function void post_randomize();
    data_rand();
  endfunction
...
endclass
```

- Constraints can control randomization of frame length
- Constraints can be attached to randomize using **with**
  - Absolute values
  - Relational constraints
- Difficult to maintain

```
randframe rframe= new("rframe");
int ok;

initial begin
  ok = rframe.randomize() with { length == 12; };
  ...
  ok = rframe.randomize() with { length > 0; length <= 7; };
  ...
```

? Question

Is there a better way to define constraints?

Constraints can be added to randomization to control the range or probability of values returned by the randomize method. Using the with construct, a constraint block, enclosed in {}, can be applied to a single randomization.

The constraint block can contain several constraint items, applied to different properties, but every item must be individually terminated with ;.

These are called in-line constraints as they are applied in-line with the randomize call. There are many different types of constraint.

Here we show absolute { length == 12; } and relational { length > 0; length <= 7;}.

If every frame has a constraint that length > 0, then it is inefficient to in-line that constraint for every randomize call. Therefore in-line constraints are difficult to maintain. A better approach is to define constraints in the classes themselves.

Class Constraints

```
class constraintframe;

      local bit[3:0] addr;
 rand local bit[3:0] length;
 rand       bit[7:0] payload[];

...

 constraint frame_length {
    payload.size() == length; }

 constraint min_length { length > 0; }

 ...
endclass
```

No ; at end of constraint line

```
constraintframe cframe;
int ok;

initial begin
  cframe = new("cframe");
  ok = cframe.randomize();
  ...
  ok = cframe.randomize() with { length == 12; };
...
```

- Constraints can be declared as class properties
- Constraints can enforce dependencies for randomization
  - Size of dynamic array must equal `length` property
- Dynamic array is now `rand`
  - Both size and contents are randomized, but...
  - ...size is randomized first
- `post_randomize` and `data_rand` methods are no longer needed
- In-line constraints no longer needed
  - But can be used if required...

14

Constraints can be defined as class properties using the constraint keyword and named. Use meaningful names for readability. Remember one constraint class property can define many constraint terms, and every constraint item within the constraint block must be terminated with a semi-colon. However, in conflict with the normal rules of Verilog, the constraint declaration itself should not be terminated with a semi-colon (although some compilers allow this for consistency).

Remember that individual constraints can be turned on and off, so partition of constraints into individual constraint blocks needs to be done with care. For example, keep unrelated constraints separate.

Constraints can be used to ensure all randomized class instances are legal. Here constraint min_length ensures that randomization will not create a frame of zero length. A class constraint is more efficient and robust than passing an in-line constraint to every randomize call.

The constraint can also define dependencies between randomized properties. Here constraint frame_length defines the number of bytes in payload must be equal to length. Remember every dynamic array variable has the method size() defined automatically. This constraint now allows us to randomize payload as well as length .When dynamic array's are randomized, the size of the array is randomized first, and the contents of the array next. This allows us to remove the post_randomize and data_rand methods of the previous example and simplify the class.

In-line constraints can still be used to control randomization, although there is an increased risk of conflicting constraints leading to randomization failure.

More Constraints : inside and dist

- `inside` defines list of values for randomization
  - Can include individual values and ranges
  - Each value in list is equally likely (uniform distribution)
- `dist` defines list of values with different probabilities
  - Called weights (default 1)

```
constraintframe cframe;
int ok;

initial begin
  cframe= new("cframe");
  ...
  ok = cframe.randomize() with {length inside {2,[6:8]}; };
  ...
  repeat (3) begin
    ok = cframe.randomize() with {length dist {[1:7],[8:15]:=2}; };
    ...
  end
...
```

Values 2,6,7,8 with uniform distribution

Values 1-15 with user-defined distribution
Value in range 8-15 twice as likely as 1-7

? What is the probability of 7 in the `dist` constraint?

Question

15

The inside operator is a convenient way for defining a list of possible values from which randomization will select. Be careful with syntax. The inside list is enclosed in {} and consists of individual values and ranges separated by commas. Each value in the list is equally likely (uniform distribution).

To change the uniform distribution, we can use dist constraints. This is similar to the inside constraint, except that values or ranges can be assigned a relative probability, or weight ,making them less or more likely to be generated in randomization. The default weight is 1. Here the weight of values in the range 1 to 7 are, by default, 1. Values in the range 8 to 15 each have a relative weight of 2.,making them twice as likely to be generated than values in the range 1 to 7. The probability of an individual item can be calculated by dividing it's individual weight by the total sum of weights. i.e. probability of value 8 is 2/((7*1)+(8*2)) = 2/23 ~9%. Compare this to uniform distribution where the probability of 8 is 1/15 ~ 7%.

Conditional Constraints

```
typedef enum {ANY, MIN, SML, MED, LGE, MAX} length_mode_t;

class modeframe;
  ...
            bit[3:0] addr;
  rand local bit[3:0] length;
  rand       bit[7:0] payload[];
  rand       length_mode_t mode;
...
 constraint frame_length {payload.size() == length; }

 constraint lengthmode
    {mode == MIN -> length == 1;
     mode == SML -> length inside {[2:5]};
     mode == MED -> length inside {[6:10]};
     mode == LGE -> length inside {[11:14]};
     mode == MAX -> length == 15; }
...
```

Declare mode enum external to class

- Constraints could be conditional
- Controlled by a mode
  - Defined with an enumerate for readability
  - Has unconstrained value
- User constrains mode which constrains length

```
modeframe mframe;
int ok;

initial begin
  mframe = new("mframe");
  ok = mframe.randomize();
  ...
  ok = mframe.randomize() with { mode == SML; };
  ...
```

enum values visible outside class

? Question

What is the probability of a `MIN` frame if `mode` is unconstrained?

16

Constraints can be conditional and the conditions can be controlled by another class property.

Here the property mode controls the constraints on length (and could also constrain other properties).

For readability, mode is defined with an enumerate type which is declared external to the modeframe class. This allows visibility of the enumerate values outside the class so they can be used with in-line constraints. So the user can set mode with in-line constraints and control the randomization of the frame. mode also has a value ANY which leaves length unconstrained.

However in the case where a randomized property controls the randomization of other properties, you may not see the results you expect. For example, what do you think is the probability of a MIN frame if mode is unconstrained?

You may assume it is 1/6, but you would be wrong...

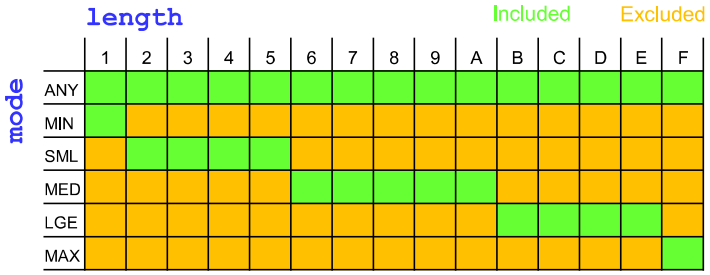
Ordering and the Constraint Solution

Unordered solution:

- `length` and `mode` randomized simultaneously
- Constraints applied
- All solutions equally likely
- Probability `MIN` is 1/30

```
constraint lengthmode { mode == min -> length == 1; ...}
```

Conceptual un-ordered solution

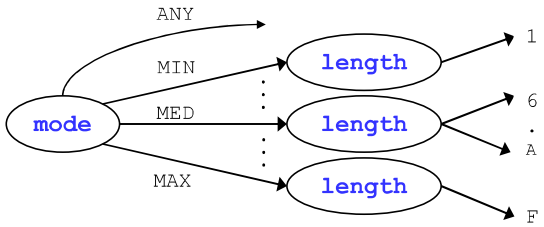


Ordered solution:

- `mode` randomized first
- `length` randomized based on constraints
- Probability `MIN` is 1/6

```
constraint lengthmode { mode == min -> length == 1; ...
                        solve mode before length; }}
```

Conceptual ordered solution



17

In the un-ordered solution, both mode and length are randomized simultaneously. This generates 15x6 = 90 possible combinations for mode and length. Then the constraints are applied and the invalid combinations are removed, e.g. when mode is MIN, any value of length other than 1 is invalid. The remaining solutions are all equally likely, meaning the combination of mode = MIN and length = 1 has a probability of 1/30.

In the ordered solution, the solve constraint requires the simulator to generate mode before length. This gives a 1/6 chance of mode = MIN, from which the only possible value of length is 1. As mode is generated first, the combination of mode = MIN and length = 1 has a probability of 1/6 .

These are conceptual explanations to aid understanding. Optimizations in the Random Number Generation may use different processes, but will generate the same results.

Randomization Failure

randfail.sv

```
module randfail.sv
  class frame;
    rand local bit[3:0] length;
    constraint min_length { length > 0; }
    ...
  endclass

  int ok;
  frame cframe = new("cframe");

  initial begin
    ok = cframe.randomize() with {length == 0;});
    ...
```

Declarative constraint

Procedural constraint

Conflicting constraints will fail randomization

Failure gives a warning, identifying:

- Conflicting constraints
- Variables used in constraints

If any one constraint fails, no randomization occurs

Randomization failure warning

xrun.log

```
 ok = cframe.randomize() with {length == 0;};
                          |
xmsim: *W,SVRNDF (./randfail.sv,...): The randomize method call failed...
...
xmsim: *W,RNDOCS: These constraints contribute to the set of conflicting constraints:

 constraint min_length { length > 0; } (./randfail.sv,4)
  ok = cframe.randomize() with {length == 0;}; (./randfail.sv,47)
xmsim: *W,RNDOCS: These variables contribute to the set of conflicting constraints:

rand variables:
       length [./randfail.sv, 3]
```

Conflicting constraints

Variables used in conflicting constraints

18

Randomization will fail if a solution cannot be found which meets all applicable constraints. By default, the failure will generate a warning message, not an error or fatal message. The message identifies the conflicting constraints and the variables used in the constraints. If the list of variables includes static (non-random) variables, then their value at the time of randomization will be shown.

The actual format of the message may vary between tool versions but the essential information is always included.

If randomization fails for one variable, then it fails for all variables used in the randomize call, and no values are changed.

In batch-mode, by default, the simulation generates the warning and continues. Therefore you need to check the simulation logs carefully for failures.

In the GUI, by default, the simulation generates the warning; stops the simulation and opens the Constraints Debugger Window showing the information on the conflicting constraints and variables affected.

Compilation Conventions

frame.sv

```
typedef enum ...

class frame;
              bit[3:0] addr;
  rand local  bit[3:0] length;
  rand        bit[7:0] payload[];
...
```

frame_pkg.sv

```
package frame_pkg;

  `include "frame.sv"

endpackage
```

frametest.sv

```
module frametest;

  import frame_pkg::*;

  frame frm;
  int ok;

initial begin
  frm = new("frm");
  ok = frm.randomize();
  frm.print();
...
```

- Each class typically declared in separate file
- Related files are included into packages
- Packages are imported into test modules
- Packages and test modules are compiled from run files
  - incdir is used to resolve includes from other directories
- Run files are accessed via a simulator option

run.f

```
-incdir ../sv
../sv/frame_pkg.sv
frametest.sv
```

```
%xrun -f run.f
```

19

For maintenance and portability, it is a convention that most SystemVerilog classes are declared in separate files. However, the class files cannot be compiled directly, but can only be compiled from a module or package scope.

The recommendation is to include related class files into a package, and then import the package into the module where it is required. Remember that the package file must be separately compiled on the command line. For convenience, because we typically have a large number of compile files and compilation and simulation options, we use run files to list files and options.

The run file also allows us to use incdir compilation options. A package can be compiled from a different directory by prefacing the file with a relative pathname, as shown for frame_pkg.sv. However the includes of the package will be resolved from the compilation directory. To allow the compiler to find the included files in the remote directory, we add incdir options to compilation which define search directories for the resolution of included files. Note you should never add the pathname to the include as this forces a directory structure and breaks reusability.

Class Review Quiz

1. Given the following constraint, what is the probability of:

```
ok = cframe.randomize() with {length dist {1,[2:5]:=2,[6:10]:=3,[11:14]:=2,15};};
```

   a) 9
   b) 1
   c) 12

2. In the code to the right, randomization fails due to conflicting constraints. What is the output of the `$display` statement:

Solutions at end of module

```
class frame;
  rand bit[3:0] addr;
  rand bit[7:0] data;
  constraint addr_not0 {addr != 0;}
endclass

int ok;
frame frm;

initial begin
  frm = new();
  frm.addr = 5;
  frm.data = 8'hf;
  ok = frm.randomize with {addr == 0;};
  $display("addr %0d data %0d",frm.addr,frm.data);
...
```

20

This page does not contain notes.

Class Review Quiz Solutions

1. Given the following constraint, what is the probability of:

```
ok = cframe.randomize() with {length dist {1,[2:5]:=2,[6:10]:=3,[11:14]:=2,15};};
```

   a) 9
   b) 1
   c) 12

2. In the code to the right, randomization fails due to conflicting constraints. What is the output of the `$display` statement:

See notes

```
class frame;
  rand bit[3:0] addr;
  rand bit[7:0] data;
  constraint addr_not0 {addr != 0;}
endclass

int ok;
frame frm;

initial begin
  frm = new();
  frm.addr = 5;
  frm.data = 8'hf;
  ok = frm.randomize with {addr == 0;};
  $display("addr %0d data %0d",frm.addr,frm.data);
...
```

21

1) Probability of a value in a dist constraint is the individual weight of the value divided by the total number of weights in the dist. Remember that the default weight is 1. Remember also for a range weight using := , each value in the range is given the defined weight. Therefore total number of weights is 1+ (4x2) + (5x3) + (4x2) + 1 = 33.

Probability of value 9 is 3/33 = 1/11 = ~9.1%

Probability of value 1 is 1/33 = ~ 3%

Probability of value 12 is 2/33 = ~ 6.1%

2) The randomize will fail on the conflicting addr constraints. If any constraint conflicts, then randomization fails and the class instance is left unchanged. Therefore the $display will show the class property values from before the randomization and the output is:

addr 5 data 15

Overview of DUT

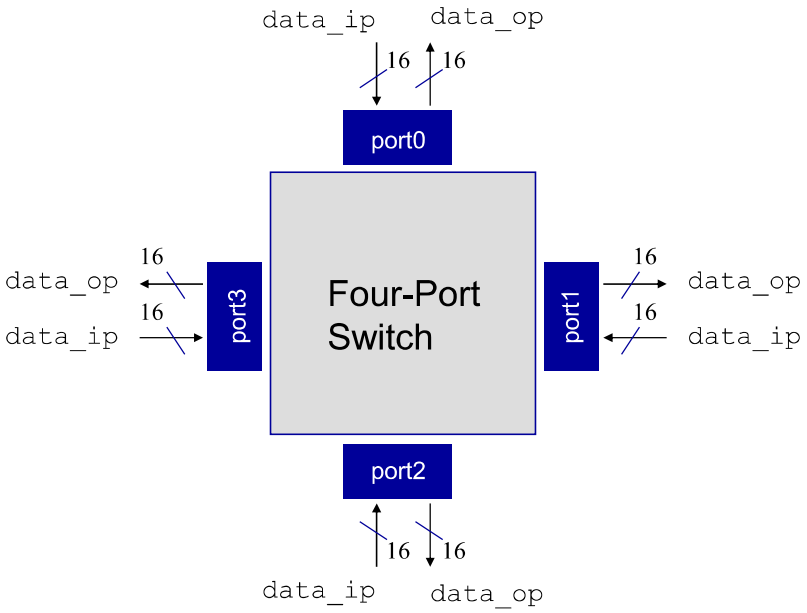
Module	**3**
Revision	3.0
Version	1.2.5

Module Objective

In this module, you

- Understand the specification of the 4-port switch design

DUT: Four Port Switch



Four port packet switch

- Packets received on all 4 ports
- Packets output on one or more port(s) according to its target address

24

We will be using the Four-Port switch as the DUT for the lab exercises throughout this course.

The switch has four ports which contain both input and output signals. The switch accepts data packets on all ports and routes the packets to one or more of the switch ports depending on the target address of the packet.

Switch Packet Specification (Simplified)

Switch packet has two bytes: Address and Payload

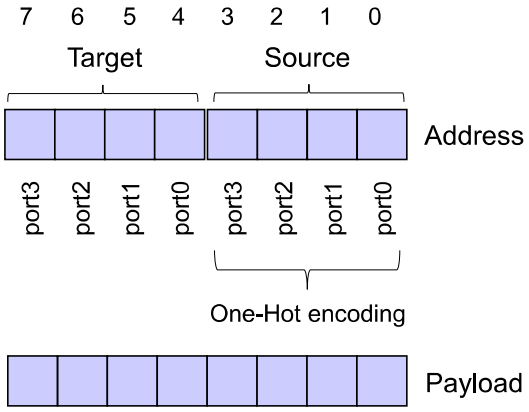
Address has two four-bit fields: Target and Source

- Source identifies input port of packet with One-Hot encoding
  - e.g. Source `4'b0001` was input at port0
- Target identifies destination port
  - e.g. Target `4'b1001` is sent to port3 and port0

Payload is 1 byte

There are 3 packet types:

1. Single sent to any 1 port *other* than Source
   - i.e. same bit cannot be set in both Target and Source
2. Multicast sent to 2 or 3 ports *other* than Source
   - i.e. same bit cannot be set in both Target and Source
3. Broadcast sent to every port (including source)
   - i.e. Target is `4'b1111`



25

A (simplified) switch packet has two bytes, address and payload.

The address byte has two fields, each of four bits.

The source field (bits 0-3) identifies the port where the packet entered the switch using a one-hot encoding. i.e. bit0 represents port0, bit1 = port1, bit2 = port2 and bit3 = port3.

The target field identifies the port(s) to which the packet is sent. Target uses the same encoding as source, but does not have to be one-hot since one input packet can be sent to multiple output ports, depending on the packet type (see below).

In the simplified switch, the payload data is one byte.

There are three packet types, identified by the target value.

A single packet is sent to a single output port. However this port cannot be the source port, i.e. a single packet cannot be sent to the same port that input the packet. For a single packet, a single bit is set in target and this bit cannot be set in source.

A multicast packet is sent to either 2 or 3 output ports. However the output ports cannot include the source port, i.e. a multicast packet cannot be sent to the same port that input the packet. For a multicast packet, either 2 or 3 bits are set in target and these bits cannot be set in source.

A broadcast packet is sent to every port, including the source. All bits are set in target.

Overview of DUT Quiz

1. What type are the following packets? Which are incorrect, and why?

```
1. 16'h0001_0001_0000_1010
2. 16'b0001_0110_0011_1111
3. 16'h0010_0001_0000_1010
4. 16'b0110_0100_1111_0000
5. 16'h0000_0100_0000_1111
6. 16'h0110_1000_0000_0000
7. 16'h1101_0010_1111_1111
8. 16'h1111_0000_1111_0101
9. 16'h1111_1000_1001_1001
```

Solutions at end of module

26

Remember the field order of the packet is target, source, payload.

Labs

Lab 1: Simple Data Class Declaration

- Properties
- Methods
- Initial testing

Lab 2: Simple Randomization and Constraints

- Constraints
- Randomization
- Initial testing
- Conditional Constraints (Optional)

27

Please consult your Lab book for details on the lab exercises.

Overview of DUT Quiz Solutions

1. What type are the following packets? Which are incorrect, and why?

1.	`16'h0001_0001_0000_1010`	: Single packet. Incorrect as same bit set in target and source
2.	`16'b0001_0110_0011_1111`	: Single packet. Incorrect as source is not one-hot
3.	`16'h0010_0001_0000_1010`	: Single packet. Correct
4.	`16'b0110_0100_1111_0000`	: Multicast packet. Incorrect as same bit set in target and source
5.	`16'h0000_0100_0000_1111`	: Unknown packet. No target bit set
6.	`16'h0110_1000_0000_0000`	: Multicast packet. Correct
7.	`16'h1101_0010_1111_1111`	: Multicast packet. Correct
8.	`16'h1111_0000_1111_0101`	: Broadcast packet. Incorrect as no source set
9.	`16'h1111_1000_1001_1001`	: Broadcast packet. Correct

28

Remember the field order of the packet is target, source, payload.

Static Properties and Methods

Module	**4**
Revision	3.0
Version	1.2.5

This page does not contain notes.

Module Objectives

In this module, you will

- Use static properties and methods

This page does not contain notes.

Static Properties

```
class staticframe;
  local string name;
  ...
  static int framecount;
  int tag;
  ...
endclass
```

Static property

Normal properties are dynamic

- Each class instance has it's own copy

A static property is shared by all class instances

- Writing to a static property from one instance changes it's value for all instances

```
staticframe f1, f2;

initial begin
  f1= new("f1",4);
  f2= new("f2",5);

  f1.framecount = 15;
  $display("f2 count %0d", f2.framecount);
...
```

Writing from one instance...

... updates for all instances

```
f2 count 15
```

31

Normally each class instance has a separate, unique copy of every class property.

However a property can be defined as static, which means that only one copy of the property exists and this copy is shared amongst all instances of the class. An update to a static property from one instance updates the property for all instances.

Here we have a static property framecount which will be used to keep track of the number of frames created, and also to assign a unique identity to each frame.

We create two instances of staticframe in handles f1 and f2. Both instances have separate copies of the dynamic property tag, but share the same static property framecount. Any changes to a static property from one handle are visible from all handles of the class. When we set framecount to 15 using the f1 handle instance, f2.framecount is also set to 15.

Conceptual Static Property Implementation

```
class staticframe;
  local string name;
  ...
  static int framecount;
  int tag;
  ...
endclass
```

Conceptually, static properties are:

- Allocated in memory at elaboration
- Linked in to each class instance at construction

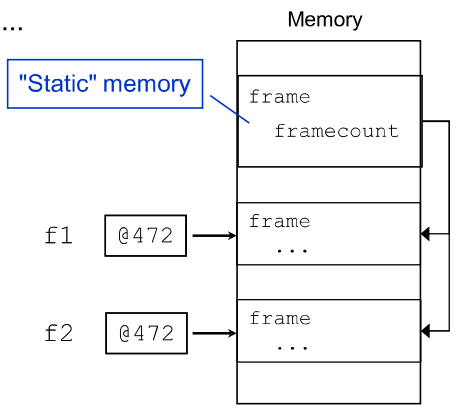
Therefore static properties can be accessed on null handle

- However this may be considered bad practice
- Better to use a static method...

```
staticframe f1, f2;
int count;

initial begin

  $display("count %0d", f1.framecount);

  f1= new("f1",4);
  f2= new("f2",5);
...
```

Null handle access

count 0



32

Conceptually static properties are created during elaboration in special areas of memory. Then every time you construct an instance of the static property class, the static property is linked into the new class instance.

As the static property is created in elaboration, we can access the property from a null handle, i.e. a handle of the static property class in which an instance has not yet been constructed and which therefore contains the value null. For example, here we access the static property framecount from the handle f1 before the constructor for f1 is called.

This is usually considered as bad practice for readability. With this option, there is no way to distinguish dynamic from static properties. Therefore a user may assume an instance already exists in f1 and attempt to access a non-static property which would give runtime errors.

We will look at better options, such as static methods and resolution access, over the next slides.

Static Methods

```
class staticframe;
  ...
  static int framecount;
  int tag;

  static function int get_count();
    return (framecount);
  endfunction
  ...
```

Can only access static properties or other static methods

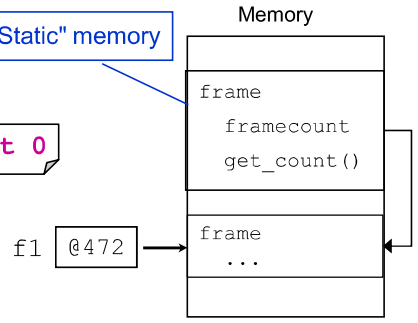
Can also be called even if no class instantiations exist

- From any class handle
  - Can be considered bad practice
- From class name using resolution operator ::
  - Best practice as clearly identifies member as static

```
staticframe f1;
int count;

initial begin
  $display("count %0d", f1.get_count());

  $display("count %0d", staticframe::get_count());

  f1= new("f1",4);
...
```

Null handle access

Resolution operator access



33

A static method can only access static properties or other static methods. Again a static method in created during elaboration in a special area of memory alongside static properties.

As with static properties, a static method can be called even if there are no current instances of the class. There are two ways to achieve this:

A static method can be accessed from any handle of the class, regardless of whether the handle contains an instance, as with a static property.

A static method can be called from the class name using the resolution operator (::)

<class_name>::<static_method>

Some coding guidelines mandate the calling of static methods using resolution operator access from the class type name. This serves to clearly indicate the method is static, and aids readability. For example, at first glance the get_count() call directly from f1 could be interpreted as a nonstatic method call from an object instance, and imply (incorrectly) that an instance exists in f1. The resolution operator syntax is unambiguous as a static method call.

This example uses a static function to return the static property framecount.

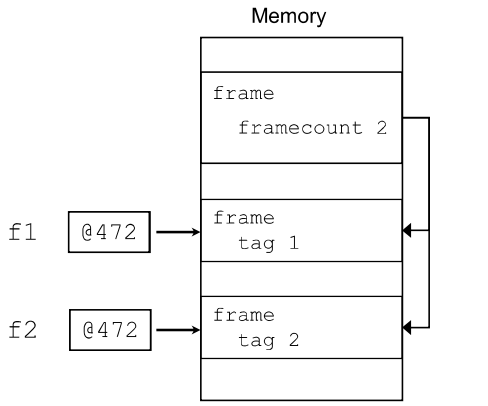
Static Properties Example

```
class staticframe;
  local string name;
        bit[3:0] addr;
  ...
  static int framecount;
  int tag;

 function new(string name, bit[3:0] addr);
   this.name = name;
   this.addr = addr;
   framecount++;
   tag = framecount;
 endfunction : new

endclass
```

```
staticframe f1, f2;

initial begin
  f1 = new("f1",4);

  f2 = new("f2",5);
...
```

Increment static `framecount` in constructor and assign to `tag`

- Counts number of frames created
- Create a unique identity `tag` for each instance



34

In this example we use the static property framecount to keep track of the number of frames created, and also to assign a unique identity to each frame. framecount is incremented in the class constructor and assigned to the nonstatic property tag.

When staticframe instance f1 is created, framecount is incremented to 1 and assigned to f1.tag. When instance f2 is created, framecount is incremented to 2, which sets both f1.framecount and f2.framecount to 2, and f2.tag is assigned to 2.

Static property framecount initializes to 0 (being of type int), but could also be explicitly initialized in the declaration:-

static int framecount = 0;

Note that we cannot initialise framecount in the class constructor. The aim is to keep an independent count of the number of frames created, so every time we create a frame, we want to increment the current frame count, not set it to zero.

tag could be used in the data item, for example, set to the first location of the payload in a post_randomize() call. Then we have a unique frame identifier that could stay with the frame as it is processed by the DUT, be recovered at the output and, for example, help match frames in a scoreboard. However here may be many other issues to consider to implement this correctly. For example, an 8-bit payload will only give us 256 unique tags. We may need to implement a scheme to reuse matched tags from the scoreboard.

Static Properties and Methods Discussion

If you wanted to keep a count of the *current* number of instances of a class, not the total number constructed, how could you implement this?

Why would you *not* want to do this?

```
class staticframe;
  local string name;
        bit[3:0] addr;
  ...
  static int framecount;
  int tag;

 function new(string name, bit[3:0] addr);
   this.name = name;
   this.addr = addr;
   framecount++;
   tag = framecount;
 endfunction : new

endclass
```

Solutions at end of module

35

This page does not contain notes.

Labs

Lab 3: Static Properties and Methods (Optional)

- Adding static properties and methods to the packet class
- This lab is optional and is not required for the creation of the 4-port switch test environment

36

Please consult your Lab book for details on the lab exercises.

Static Properties and Methods Discussion

If you wanted to keep a count of the *current* number of instances of a class, not the total number constructed, how could you implement this?

Why would you *not* want to do this?

See slide notes

```
class staticframe;
  local string name;
        bit[3:0] addr;
  ...
  static int framecount;
  int tag;

 function new(string name, bit[3:0] addr);
   this.name = name;
   this.addr = addr;
   framecount++;
   tag = framecount;
 endfunction : new

 static function void deconstruct(inout staticframe frm);
    framecount--;
    frm = null;
  endfunction

endclass
staticframe frm1;
...
  staticframe::deconstruct(frm1);
```

`inout` as the method executes a read/modify/write on the argument

User-defined deconstructor to decrement `framecount` and `null` the handle

Deconstructor call

37

One solution would be to add an explicit de-constructor. This is declared as a static method of the class. It would be called whenever an instance is not longer required. The de-constructor decrements framecount and also makes sure the instance is no longer used by assigning it to null in an attempt to recycle the instance.

The disadvantage is that it is virtually impossible to guarantee an accurate count:

SystemVerilog does not use de-constructors. Memory management is automatic. There is no way to force a user to recycle instances using an explicit de-constructor, therefore the count would not be reliable.

Assigning a handle to null does not necessarily recycle the instance memory. Another handle may be pointing to that instance (because you copied the handle). The count is decremented but the instance is still in use, leading, again, to an unreliable count.

An accurate count would require a C-like manual recycling of instances. This would be very hard to enforce on top of the automatic recycling of SystemVerilog, so is probably doomed to failure.

Inheritance and Polymorphism

Module **5**

Revision 3.0
Version 1.2.5

Module Objectives

In this module, you will

- Use inheritance and polymorphism:
  - Inheritance for constraint layering
  - Constructors in inheritance
  - Polymorphism
  - Virtual Methods

39

This page does not contain notes.

Simple Inheritance Review

```
class frame;
                bit[3:0] addr;
  rand protected bit[3:0] length;
  rand           bit[7:0] payload[];

 function new(string name);
   this.name = name;
 endfunction

 constraint frame_length {payload.size()==length;}
 constraint min_length { length > 0; }
endclass

class smallframe_bad extends frame;
  constraint sml_length {length inside {[2:5]};}
endclass

class mediumframe_bad extends frame;
  constraint med_length {length inside {[6:10]};}
endclass
```

Base class

protected gives visibility in subclass

Subclass

Subclass

Subclasses give compilation errors

- Multi-mode classes with conditional constraints can be difficult to maintain
- Inheritance is usually better
  - New subclass for each variant
  - Extended from base class to inherit members
  - Layer constraints on those of base
  - Change `local` to `protected` for access in subclass
  - Can only extend from a single class
- Subclasses can be added at any time without affecting original data class
- Remember constructors are not inherited

Question

What is the problem with **smallframe_bad** and **mediumframe_bad** ?

40

Generating different frame types with mode switches can be difficult to maintain.

Inheritance is usually easier, more robust and reusable. We create a new subclass for every frame variant, extended from the original frame class to inherit all the class members. Then we can simply declare constraints in the subclass to create the required frame. Note that subclass constraints cannot access local properties in the parent class as these are only visible in the parent. You need protected properties which are visible in the parent class and any subclasses.

Remember SystemVerilog only supports single inheritance, i.e. each subclass can only extend from a single parent. SystemVerilog 2012 introduced Java-like interface classes which allow a form of multiple inheritance. However these are rarely used as the original UVM class library pre-dates SV2012 and so does not support interface classes.

A user can add a subclass at any time without having to edit the original data class, adding flexibility.

Remember that constructors are not inherited. Therefore the smallframe_bad and mediumframe_bad subclasses will fail compilation. Why is this? See the next slide for the answer.

Inheritance and Constructors

```
class frame;
 ...
 function new(string name);
   this.name = name;
 endfunction
 ...
endclass
```

```
class smallframe_bad2 extends frame;

  function new();
    super.new();
  endfunction

  constraint sml_length {length inside {[2:5]};}
endclass
```

Default constructor: compilation error

```
class smallframe extends frame;

  function new(string name);
    super.new(name);
  endfunction

  constraint sml_length {length inside {[2:5]};}
endclass
```

Constructors are not inherited

- A default constructor is added
  - This gives a compilation error – why?
- An explicit constructor overrides the default
  - A default super call is added as first line
    - `super.new();`
  - This gives a compilation error – why?
- An explicit `super` call as first line of explicit constructor overrides the default super call
- Constructors should be explicit and cascaded

41

Constructors are not inherited by subclasses.

Without an explicit constructor, a default constructor is automatically added, as in smallframe_bad2. This gives a compilation error, as the super.new() call of the default constructor is not passing a value to the name argument of the frame constructor.

Adding an explicit constructor will overwrite the default implementation. However a default super.new() call is added as the first line of any explicit constructor, which will give the same compilation error as before. Adding an explicit super.new call as the first line of an explicit constructor will overwrite the default implementation and allow the user to pass an argument value to the frame constructor.

So constructors should always be explicit, with an explicit super.new call as the first line.

As super.super.new calls are not allowed, in an inheritance hierarchy with more than two levels, constructor argument values must be passed up one level at a time, i.e. cascaded.

Constraint Block Inheritance

```
class frame;
 ...
 constraint frame_length {payload.size() == length; }
 constraint min_length { length > 0; }

endclass

class smallframe extends frame;
   ...
  constraint sml_length {length inside {[2:5]};}
endclass

class uppersizeframe extends frame;
  ...
  constraint min_length {length > 7;}
endclass

class illegalframe extends frame;
  ...
  constraint min_length {}
endclass
```

Layer

Override min_length

Remove min_length

Constraints are class members and are inherited just like other members

With subclasses, constraints can be:

- Layered
  - Add new constraint on top of base
- Overridden
  - Redefine base constraint
- Removed
  - Redefine base constraint as empty

Create instances of subclasses and randomize

42

Constraints are class members and are inherited just like all other inherited class members.

Our frame class has two constraints, named frame_length and min_length.

Subclass smallframe defines a new constraint sml_length, which layers on top of the frame constraints. When instances of smallframe are randomized, local constraint sml_length and inherited constraints frame_length and min_length all apply and length is restricted to the range 2,3,4,5 with payload size equal to length.

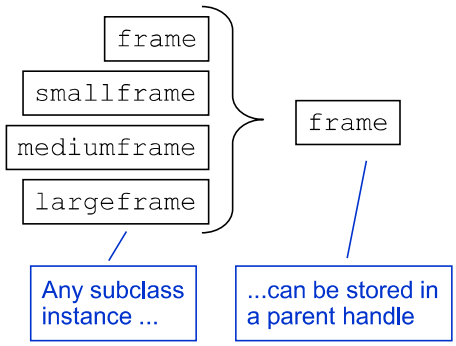
Subclass uppersizeframe redefines the constraint min_length, which overrides the frame constraint of the same name. When instances of uppersizeframe are randomized, local constraint min_length and inherited constraint frame_length both apply and length is restricted to the range 8 to max(15) with payload size equal to length.

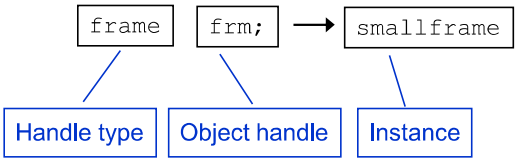
Subclass illegalframe redefines the constraint min_length, to be empty and overrides the frame constraint of the same name. When instances of illegalframe are randomized, only inherited constraint frame_length applies and so length could be randomized to 0.

In cases such as uppersizeframe when you want to redefine an existing constraint, it may be confusing to use the same name. It may be better to remove the existing constraint (as in illegalframe) and then define a new constraint with a more meaningful name.

We can now create instances of each subclass and randomize to pick up the required constraints.

Polymorphism





A subclass creates a new type

You need to easily change between subclass types

- Example: switch to `smallframe` without updating type of all object handles

A class handle of a given type can be assigned any inherited class instance

- Polymorphism

Introduces the concept of a handle *type*

- Class type is used in declaration

... and the *contents* of the handle

- Class instance is held by the handle

43

Every subclass declaration defines a new type, e.g., frame, smallframe, mediumframe and largeframe are all different type declarations in SystemVerilog. However smallframe, mediumframe and largeframe are all subclasses of frame.

We need to easily change between different subclass types without having to change the type of our class handles. For example, if the testbench environment was written using frame handles, we don't want to edit these handles to run the testbench with smallframe, or a mix of small, medium and large frames.

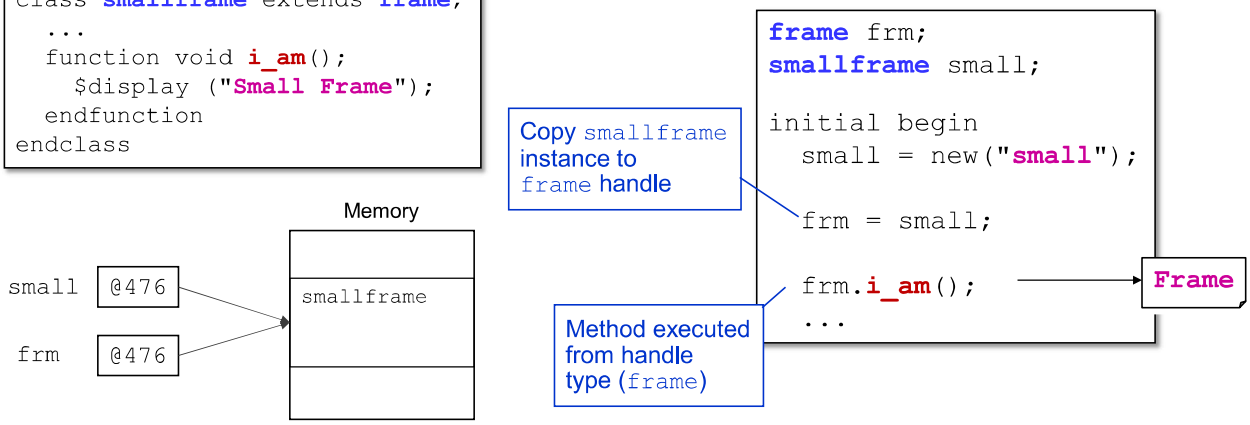
A key feature of object-oriented design is the ability to assign to a class handle of a specific type, any instance of a subclass type. i.e. we can create our design using object handles of type frame, and then assign any instance of the subclasses of frame to that handle. This is polymorphism.

This introduces the concept that an object handle has a type used in its declaration, e.g. frame, but its contents – the object instance to which the handle points – may be of a subclass of the handle type, e.g. smallframe.

Assigning Subclass Instance

```
class frame;
  ...
  function void i_am();
    $display ("Frame");
  endfunction
endclass

class smallframe extends frame;
  ...
  function void i_am();
    $display ("Small Frame");
  endfunction
endclass
```

- A subclass instance can be assigned directly to a parent class handle
- However, by default, member access is resolved from the handle type
  - Even though it contains a subclass instance



44

We can always directly assign a subclass instance to a handle of a parent class type.

Note here we have added artificial methods i_am() to both frame and smallframe. This is for just clarity.

Here, we create an instance of smallframe in the handle small, and copy the instance to the frame handle frm. As frame is a parent class of smallframe, the assignment is valid. Remember that both frm and small are handle pointers to memory addresses where the instances are actually stored. Therefore when we copy small to frm, we are overwriting the current contents of frm (if any) with the address stored in small, which is a pointer to an instance of smallframe. This is called a reference copy.

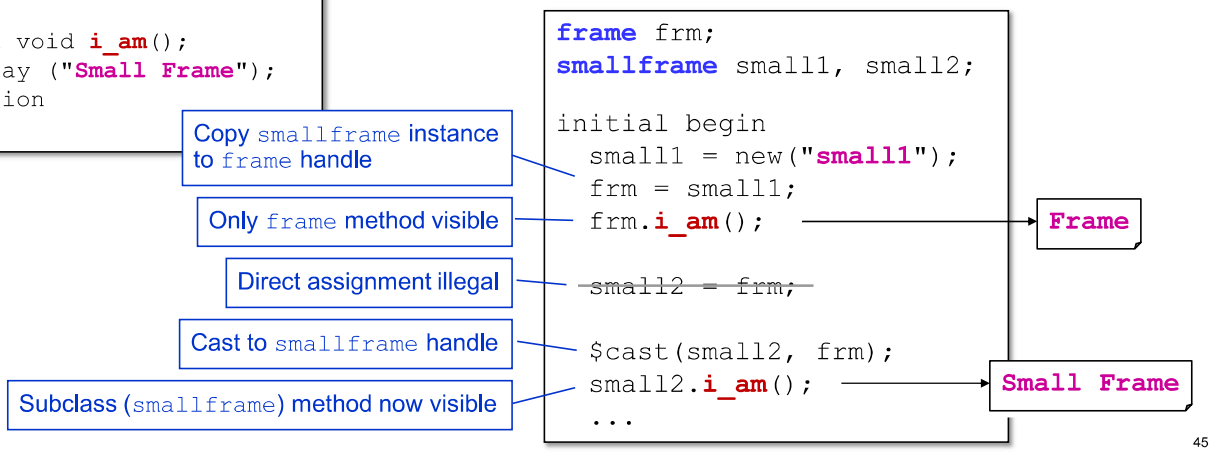
So frm has a handle type of frame, but contains (points to) an instance of smallframe.

However, by default, class member access is resolved according to the handle type. Therefore when we try to access class members from frm, we can only see frame members. Calling the over-ridden i_am() method from frm executes the function defined in the frame class, even through frm contains an instance of smallframe.

Extracting Subclass Instance

```
class frame;
  ...
  function void i_am();
    $display ("Frame");
  endfunction
endclass

class smallframe extends frame;
  ...
  function void i_am();
    $display ("Small Frame");
  endfunction
endclass
```

- A parent-class handle *cannot* be copied to a subclass handle
  - *Unless* the parent handle contains an instance of that subclass
- Direct assignment is not possible
- Requires use of `$cast`
  - Checks that the parent handle contains a subclass instance



45

One way to access the members of a subclass instance copied to a parent class handle is to copy the subclass instance back into a subclass handle.

However, although it is always legal to assign a subclass instance to a handle of a class higher in the inheritance tree, it is never legal to directly assign a parent handle to a handle of one of its subclasses. It is only legal to assign a parent handle to a subclass handle if the parent handle contains an instance of the given subclass. To check whether the assignment is legal, a dynamic cast must be used.

$cast is called with two arguments – the first is the target handle for the assignment, and the second is the source handle. $cast checks the contents of the source are compatible with the handle type of the target, and if so, it makes the assignment from source to target.

Once the subclass instance is back in a subclass handle, the correct members can be accessed.

Here we copy an instance of smallframe into the frame handle frm. When we access the method i_am() from frm the call is directed to the handle class type frame. Using the cast, we can copy the contents of frm to the handle small2. The cast checks that the contents of frm (an instance of smallframe) are compatible with the smallframe handle small2. The check is successful and so the assignment completes. We can now call the i_am() method from small2 and access the smallframe implementation.

$cast

Casting lets subclass instances use resources defined for parent classes.

- Such as handles and arguments

Syntax

`$cast (destination, source)`

`$cast` is actually a subroutine

- Defined as both function and task

If the source does not contain a matching instance for the destination:

- Task gives a run-time error
- Function returns 0

```
function frame get_next_item();
  frame  frm;
  smallframe small;
  randcase
    1: frm = new("frm");
    1: begin
       small = new("small");
       frm = small;
       end
  endcase
  return (frm);
endfunction

frame frame1;
smallframe small1;

initial begin
  frame1 = get_next_item();

  if ($cast(small1, frame1))
    $display("Small Frame");
...
```

Output argument type `frame`

Randomly create `frame` or `smallframe` instance

If small instance, copy to frame handle

Return frame handle

Get item into `frame` handle

If `$cast` succeeds, we returned a `smallframe` instance

```
Small Frame
```

46

Casting allows us to use a base or parent class resource, such as a handle or function argument, with any instance of a subclass of the base or parent type. We can assign the subclass instance directly to the base class resource, and then extract the subclass instance back into a subclass handle using the cast.

In the example we have a function get_next_item() which returns a frame instance via the return argument type. However we want the function to return smallframe instances as well. Instead of creating separate functions to return separate types, we can declare one function and assign a smallframe instance to the frame function return argument. In fact in the function, we use a randcase to randomly create a frame or smallframe instance for return, so when we call the function, we don't know which type is produced.

In the initial block, we assign the function return value to a frame handle. We then try casting the frame handle to a smallframe handle. If the casting succeeds, the function returned a smallframe instance. If the casting fails, a frame instance was created.

The syntax for $cast() is as follows:

- task $cast(destination_handle, source_handle);
- function int $cast(destination_handle, source_handle);

The function form is preferable because if the cast fails, we can detect the failure and act accordingly. The task form gives a run-time error with no possibility of recovery or debug.

Virtual Methods

```
class frame;
  ...
  virtual function void i_am();
    $display ("Frame");
  endfunction
endclass

class smallframe extends frame;
  ...
  virtual function void i_am();
    $display ("Small Frame");
  endfunction
endclass
```

By default, member access is resolved from the handle type

- Even if the handle points to a subclass instance

Virtual methods are resolved according to the *contents* of a handle

- Call methods of a subclass instance held in a parent class handle

Once a method is declared as virtual, it is automatically virtual in every subclass

Keyword `virtual` optional in subclass

Copy `smallframe` instance to `frame` handle

Method executed from handle **contents** type (`smallframe`)

```
frame frm;
smallframe small;

initial begin
  small = new("small");

  frm = small;

  frm.i_am();
  ...
```

→ `Small Frame`

47

Casting is not very flexible. If we add new subclasses to the environment, we have to add new subclass handles and cast to these. Using virtual methods is much easier.

Simply declaring a method as virtual means a call is resolved according to the contents of a handle, not the type of the handle..

Here we create an instance of the smallframe class and copy it into the frame handle frm. When we call frm.i_am(),the method call is resolved by first looking in the class type of frm, i.e., the frame class. In class frame, i_am()is declared as virtual, and this forces the resolution to go back to the call and examine the contents of frm. We find frm contains an instance of smallframe, so the resolution is directed to the smallframe class. So frm.i_am() returns "Small Frame" when i_am() is declared as virtual.

Only methods can be virtual, not properties. If you need to access a subclass property from a base class handle, you need to access the property via a virtual method.

Once a method is declared as virtual, it is automatically virtual in every subclass. Therefore declaring i_am()as virtual in frame, means it is automatically virtual in smallframe, and the virtual keyword can be omitted for the smallframe declaration. This can be considered bad practice. However as a counter-argument, you should not blindly declare every overridden subclass method as virtual. It is legal for a method to be non-virtual in the parent class, and virtual in a subclass and all it's descendants. This is by far a bigger problem. A good guideline is not to declare overridden methods as virtual if you are extending from a class library, e.g. UVM. If the method was virtual in the parent class, it is automatically virtual in the subclass. If it was not virtual in the parent class, presumably for very good reasons, you have not made it virtual in your subclass.

Inheritance and Polymorphism Quiz

```
class base;
  virtual function void one();
    $display("base1");
  endfunction

  function void two();
    $display("base2");
  endfunction

  virtual function void three();
    $display("base3");
  endfunction
endclass

class parent extends base;
  function void one();
    $display("parent1");
  endfunction

  function void two();
    $display("parent2");
  endfunction

  virtual function void four();
    $display("parent4");
  endfunction
endclass
```

Given these declarations, what is the output of the following code?

```
base b;
parent p = new();

initial begin
  b = p;
  b.one();
  b.two();
  b.three();
  b.four();
...
```

Solutions at end of module

48

This page does not contain notes.

Lab

Lab 4 Inheritance and Polymorphism

- Subclasses
- Layered constraints
- Polymorphism and virtual methods

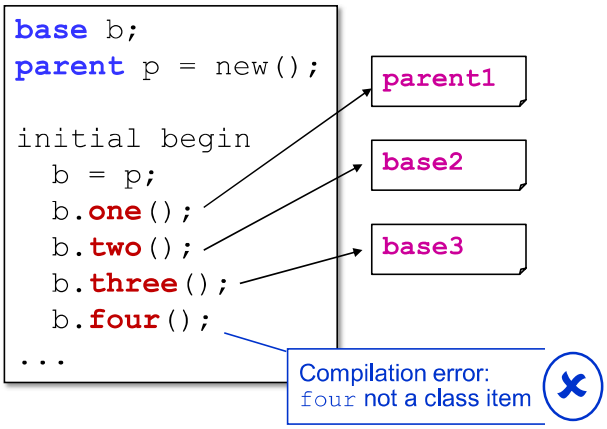
49

Please consult your Lab book for details on the lab exercises.

Inheritance and Polymorphism Quiz Solution

```
class base;
  virtual function void one();
    $display("base1");
  endfunction

  function void two();
    $display("base2");
  endfunction

  virtual function void three();
    $display("base3");
  endfunction
endclass

class parent extends base;
  function void one();
    $display("parent1");
  endfunction

  function void two();
    $display("parent2");
  endfunction

  virtual function void four();
    $display("parent4");
  endfunction
endclass
```

Given these declarations, what is the output of the following code?



50

We copy the parent instance into b, therefore b has a handle type of base but contains an instance of parent.

b.one();

The call is directed to the handle type base, where we find the method one() is virtual, so we look at the contents of b, find the instance of parent and redirect the call to the parent class where we find an implementation of one() which outputs parent1.

b.two();

The call is directed to the handle type base, where we find the method two() is non-virtual, so we execute the implementation found in base which outputs base2.

b.three();

The call is directed to the handle type base, where we find the method three() is virtual, so we look at the contents of b, find the instance of parent and redirect the call to the parent class. However the parent class does not redeclare the method three(). It inherits three() from base, so we execute the base implementation which outputs base3.

b.four();

The call is directed to the handle type base, but there is no implementation of four() in base. Therefore compilation fails with a message "four is not a class item". Even though four() is declared in the parent, the compiler always looks at the handle type first, therefore virtual classes must be declared in base classes to be used. In this case, the only was to access four() would be to $cast b to a handle of parent.

Aggregate Classes

Module	**6**
Revision	3.0
Version	1.2.5

This page does not contain notes.

Module Objectives

In this module, you will

- Review aggregate classes
- Apply common class operations to aggregate classes
  - Encapsulation
  - Reference, shallow and deep operations

52

This page does not contain notes.

Properties of Type Class

```
class frame;
  local string name;
        bit [3:0] addr;
...
  function new(string name);
    this.name = name;
  endfunction
endclass

class twoframe;
  local string name;
        frame f1;
        frame f2;
...
  function new(string name);
    this.name = name;
    f1 = new("f1");
    f2 = new("f2");
  endfunction
endclass
```

Properties of previously defined class type

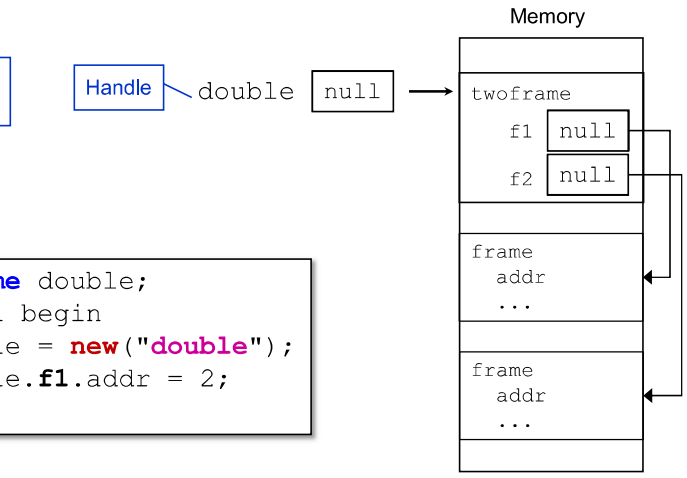
A class property can be an instance of another class

- Creates an aggregate, or composite class

Class properties must be explicitly constructed

Chain handles to reach into hierarchy

```
twoframe double;
initial begin
  double = new("double");
  double.f1.addr = 2;
...
```



53

It is common for class objects to be building blocks for other classes. One way to achieve this is to declare class properties which are themselves handles of other classes. This is known as an aggregate or composite class (an aggregation, or collection, of other class instances).

We can use this to create a class which holds multiple frames.

Here class twoframe contains two object handles (f1, f2) of the class frame as properties. The constructor for twoframe should create the instances of frame by a call to the frame constructor. Without this call the frame object handles in twoframe will be null. When we create an instance double of twoframe, the constructor creates the instances f1 and f2 of frame within the double instance.

To access the properties of f1 within double, we must chain handles,

double.f1.addr

This accesses the property addr of the frame instance f1 which is a property of the twoframe instance double.

This is similar to a module hierarchy in Verilog. We can use a hierarchical pathname to access variables in a module-based hierarchy, likewise we can use an handle pathname to access members of a class-based hierarchy.

Randomization in Aggregation

```
class frame;
  local string name;
        bit [3:0] addr;
  rand protected  bit[3:0] length;
  rand  bit[7:0] payload[];
...

endclass

class twoframe;
  local string name;
  rand  frame f1;
  rand  frame f2;
...
endclass
```

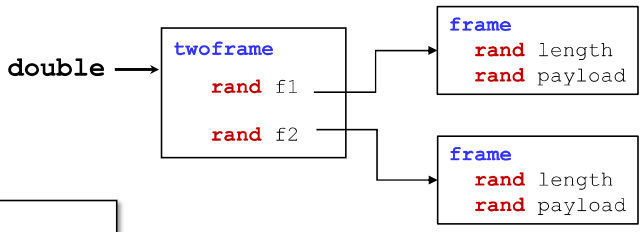
Properties of class type containing rand properties

Declare properties as `rand`

```
twoframe double;
bit ok;

initial begin
  double = new("double");
  ok = double.randomize();
...
```

Randomize can operate hierarchically on aggregate classes:

- Only if the class property is declared as `rand`
- Otherwise, that property is skipped for randomization



54

An aggregate class may contain class instance properties which themselves contain random properties. Here the twoframe aggregate class contains two instance of frame with random variables, length and payload.

The randomize method can push down into the frame instances within twoframe, and randomize length and payload, but only if the frame handles (f1 and f2) are declared as rand. If f1 or f2 is not declared as rand, properties such as double.f1.length and double.f2.payload will not be randomized, even though they are random properties.

Effectively, the rand modifier acts as a gate for randomization. If a class instance property is rand, randomization pushes into the property to randomize its contents, otherwise the instance is skipped.

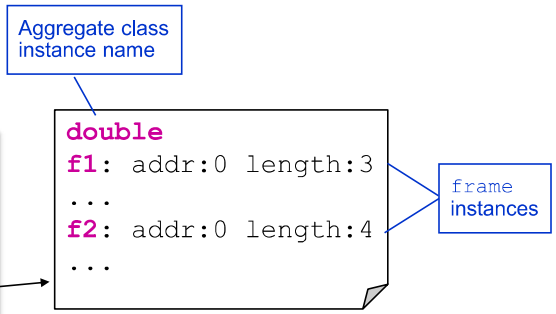
Encapsulation and Aggregate Classes

```
class frame;
  local string name;
        bit [3:0] addr;
  rand protected  bit[3:0] length;
...
  function void print();
    $display("%s: addr:%h length:%0d", name, addr, length);
    // payload print
  endfunction
endclass

class twoframe;
  local string name;
  rand frame f1;
  rand frame f2;

  function new(string name);
    this.name = name;
    f1 = new("f1");
    f2 = new("f2");
  endfunction
...
 function void print();
   $display("%s",name);
   f1.print();
   f2.print();
 endfunction
```

```
twoframe double;

initial begin
  double = new("double");
  ok = double.randomize();
  double.print();
...
```

A class is responsible for handling its own properties

- A class should print its properties
- An aggregate class calls the print methods of its class properties
- Instance names identity individual instances



55

The basic rule of encapsulation is that every class should be self-contained, and responsible for handling it's own properties.

If we consider a print function, then this means an aggregate class like twoframe should print its own local properties (name) but call the print function of f1 and f2 to print the properties of the frame instances. By including the instance name property in the print function, we can distinguish the properties of the f1 frame instance from those of the f2 frame instance.

Note: No print policy shown for simplicity.

Reference Copy

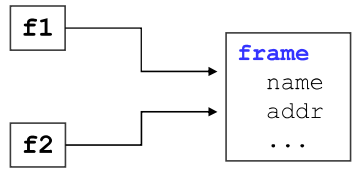
```
class frame;
  local string name;
        bit[3:0] addr;
  ...
endclass

frame f1, f2;

initial begin
  f1 = new("f1");
  ok = f1.randomize();

  f2 = f1;

  f1.addr = 7;
  $display("f2 addr %0d",f2.addr);

  f1 = null;
...
```

Reference copy

De-reference f1

f2 addr 7

- Simple copy via assignment
  - Duplicates the class instance pointer
  - Does not create a new instance
- Called a Reference copy
- A property updated via one object pointer can be seen via the other
- De-referencing one handle does not affect the second
  - Garbage collection is automatic



Reference copy

56

A simple assignment copy duplicates the class instance pointer only, creating another pointer to the same class instance in memory. This is called a Reference Copy. There is only one instance of frame, but both handles, f1 and f2, point to this instance. If we update a property value via one handle, e.g. f1.addr, then obviously this change is seen if we access the property via the other handle (f2.addr).

In classic C++ object-oriented design, reference copies can be dangerous as if one pointer is de-allocated. The other, duplicate pointer is now invalid. In SystemVerilog, de-allocation of class instances is handled by the simulator (as in Java) so this is not as big a problem as in C++.

Clone

```
class frame;
  local string name;
        bit[3:0] addr;
  rand  bit[3:0] length;
  rand  bit[7:0] payload[];
...
endclass

frame f1, f2;

initial begin

  f1 = new("f1");
  ok = f1.randomize();

  f2 = new("f2");
  f2.length = f1.length;
  f2.addr = f1.addr;
  f2.payload = f1.payload;

  f2 = new f1;
...
```

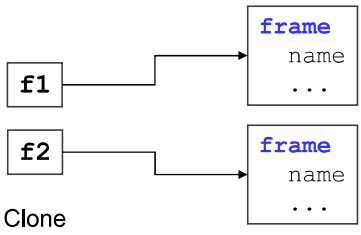
No `protected` or `local` properties for first clone

Create instance

Copy values

Easier clone

- Clone creates a new instance with the same property values
- Cloning can be done by:
  - Creating a new object
  - Copying properties from old object to new object
    - Except instance name
  - Does not work with `local` or `protected` properties
- Easier clone using new
  - `clone = new source;`
  - Not a call to the class constructor (no arguments)
  - Works with `local` and `protected` properties
    - Including instance name



Clone

57

If we want a separate copy of a class instance, rather than a reference copy, then we need a clone.

A clone creates a new instance of the class and copies property values from the original source object to the new instance. Cloning can be carried out by creating a new object using the class constructor and then separately copying every property from the original instance. You can choose which properties to copy across. For example, we can omit copying the instance name from f1 to f2.

In this example, the copy is done with procedural code. This means local and protected properties cannot be copied.

Alternatively, an easier way of simple cloning is to use the keyword new:

- target = new source ;

This is a much simpler method of cloning the source instance. All properties are copied, including the instance name, and the new clone works with protected and local properties. We may need to update the instance name after the clone to keep consistency with the handle name. This will require a method if, as here, the name is local.

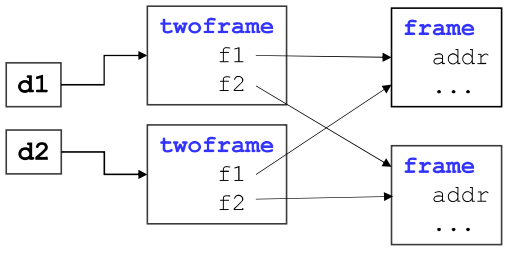
Note that new in this application is not the class constructor and so does not take any arguments.

Shallow and Deep Clone

```
class frame;
  local string name;
        bit[3:0] addr;
  rand  bit[3:0] length;
  rand  bit[7:0] payload[];
...
endclass

class twoframe;
  local string name;
  rand frame f1;
  rand frame f2;
  ...
endclas
```

```
twoframe d1, d2;
int ok;

initial begin
  d1 = new("d1");

  d2 = new d1;

  d1.f1.addr = 4'hf;
  $display("d2.f1 addr %0d", d2.f1.addr);
...
```

- Simple `new` clone is shallow
- Aggregate class properties are copied by reference
  - A new instance is not created
- Deep cloning creates new instances for all object properties
  - Creates a completely independent copy of the original class
  - Requires custom code



Shallow Clone

```
d2.f1 addr 15
```

58

The simple clone using new has an issue – it is a shallow clone. If the original class instance is an aggregate class (i.e. has a property which itself is a class instance), then the class properties are copied by reference only. A new instance of the class property is not created. So here, f1 and f2 of the double instance of twoframe are copied by reference only. New instances of f1 and f2 are not created. This means if we write to f1.addr via the d1 handle, the value also changes in the f1.addr instance of d2, as d1.f1 and d2.f1 point to the same instance.

A shallow clone refers to the fact that only the first level of the class instance hierarchy is cloned.

A deep clone duplicates the entire class hierarchy and creates an identical, but completely separate and independent copy of the original class. Deep cloning requires custom code to create new instances of class instance properties.

Deep Cloning

```
class frame;
  local string name;
        bit[3:0] addr;
  rand  bit[3:0] length;
  rand  bit[7:0] payload[];
...
endclass

class twoframe;
  local string name;
  rand frame f1;
  rand frame f2;
  ...
endclass
```

```
twoframe d1, d2;

initial begin
  d1 = new("d1");

  d2 = new("d2");
  d2.f1.addr = d1.f1.addr;
  d2.f1.length = d1.f1.length;
  d2.f1.payload = d1.f1.payload;
  d2.f2.length = d1.f2.length;
  ...

  d2 = new("d2");
  d2.f1 = new d1.f1;
  d2.f2 = new d1.f2;
...
```

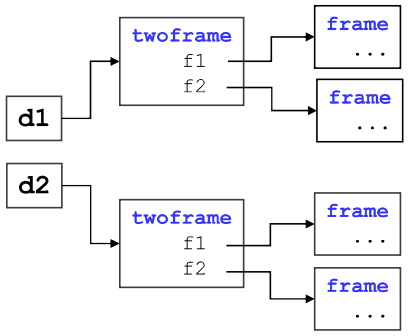
- Deep clone needs custom code
  - Create a completely new object hierarchy using constructor
  - Traverse object hierarchy, copying all properties as required
  - Can be defined as a method
- Can use `new` clone

create hierarchy

copy `f1` properties

copy `f2` properties

Or use hierarchical `new` clone



Deep Clone

59

Deep cloning requires custom code to create new instances of class instance properties. A suitable algorithm may be as follows:-

Create a new instance of the top level of the class hierarchy(e.g., doubleframe).

Push down into the first level of the class (d2).

Copy all non-class properties to the new clone (e.g. name). Here we are not copying the instance name of d1 to maintain consistency with the handle name.

Create a new instances of any class-properties at the current level (f1, and f2).

Push down into a new class property hierarchy (e.g. d1.f1) and copy all the properties across (payload, length, address)

Repeat steps 4-5 until no more class-properties are found.

Pop back up the class hierarchy, checking at each level for further class properties to be cloned, and repeating steps 4-6 for each.

We can use the new clone for leaf instances, i.e. instances which do not contain further properties of class types.

Aggregate Classes Quiz

1. Given the following declarations and code, what would be the result of the compare if it was:
   a) A reference compare
   b) A shallow compare
   c) A deep compare
2. What difference (if any) would it make if the comment line was uncommented?

```
class sub;
  bit subprop;
endclass

class twosubs;
  int count;
  sub sp1,sp2;

  function new();
    sp1=new();
    sp2=new();
  endfunction
endclass
```

```
twosubs tsa, tsb;

initial begin
  tsa = new();
  tsb = new tsa;

  //tsa.count = 1;

  tsa.sp1.subprop = 1'b1;

  <compare tsa and tsb>
...
```

Solutions at end of module

60

This page does not contain notes.

Aggregate Classes Quiz Solution

1. Given the following declarations and code, what would be the result of the compare if it was:
   a) A reference compare - fail
   b) A shallow compare – pass, fail when line uncommented
   c) A deep compare – pass, fail when line uncommented
2. What difference (if any) would it make if the comment line was uncommented?

```
class sub;
  bit subprop;
endclass

class twosubs;
  int count;
  sub sp1,sp2;

  function new();
    sp1=new();
    sp2=new();
  endfunction
endclass
```

```
twosubs tsa, tsb;

initial begin
  tsa = new();
  tsb = new tsa;

  //tsa.count = 1;

  tsa.sp1.subprop = 1'b1;

  <compare tsa and tsb>
...
```

Shallow clone

61

Note that the properties subprop and count are both bit based and so have default starting values of 0.

tsb = new tsa; is a shallow clone. Therefore the sub handles in both tba and tsb point to the same instances in memory. Assigning tsa.sp1.subprop to 1, also assigns tsb.sp1.subprop.

A reference compare will fail as the tsa and tsb handles have different values (point to different areas of memory). This is because the clone created a new instance for tsb.

A shallow compare will pass. The clone copies the value of count, and the sp1 and sp2 handles of both tsa and tsb point to the same instances in memory (because we used a shallow clone).

A deep compare will pass. The count and sp1.subprop and sp2.subprop values are the same in both tsa and tsb. As described above, assigning tsa.sp1.subprop to 1 also changes tsb.sp1.subprop to 1 because we used a shallow clone.

If we uncomment the tsa.count = 1; line, then tsa.count is different from tsb.count (which has the default value of 0). Therefore both shallow and deep clones will fail if the line is uncommented.

Components

Module 7

Revision 3.0
Version 1.2.5

This page does not contain notes.

Module Objectives

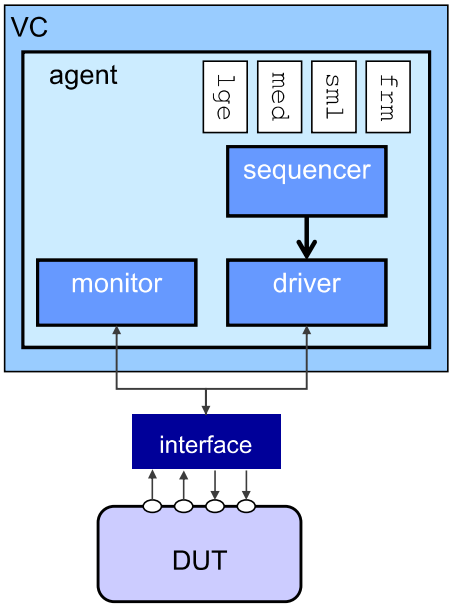
In this module, you will

- Create component hierarchies using aggregate classes
- Use:
  - Component hierarchy
  - Instance names and parent pointers for hierarchical navigation
  - Connect components using references

63

This page does not contain notes.

Architecture for Driving Data into DUT



Components required to drive data into DUT:

- Sequencer
  - Generate and randomize data items
- Driver
  - Converts data item into signal transitions
  - Drives interface signals with correct protocol
- Monitor
  - Capture signal transitions on interface and regenerate data
- Agent
  - Encapsulate sequencer/driver/monitor
  - May have multiple agent instances
    - e.g. array of identical interfaces
- Verification component (VC) top level
  - Allows for easy reuse
- More details in following modules...

64

We need a Verification Component to drive our data items into the DUT. Convention, and UVM, use the following architecture.

A sequencer class generates the data instances and performs randomization.

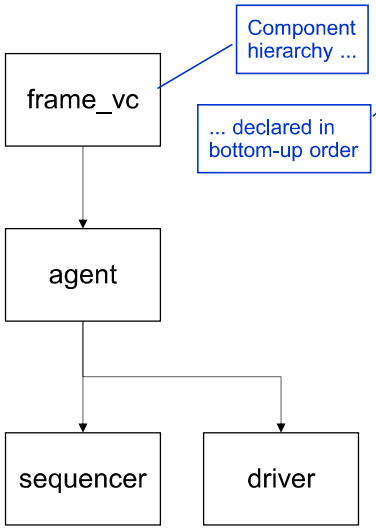
The instances are passed to a driver class which implements the protocol to convert each data instance into signal transitions on the ports of the DUT, via a SystemVerilog interface.

A monitor class captures the signal transitions on the interface signals and regenerates the data items sent into the DUT. The monitor can them perform checking and coverage and pass the data onto (for example) scoreboards.

The sequencer, driver and monitor classes are instantiated as properties in an agent class. There may be several different types of agent in a verification component, for example transmit and receive agents for driving both side of a simple bus protocol. Agent(s) are instantiated in a final class layer which is the top level of the verification component.

All the classes described here are component classes. They have special properties and methods beyond those declared in a data item as we shall see.

Verification Component Hierarchy



Component hierarchy ...

... declared in bottom-up order

```
class sequencer;
  ...
endclass

class driver;
  ...
endclass
...
class agent;
  driver drv;
  sequencer seqr;
  ...
  function new(...);
    drv = new(...);
    seqr = new(...);
  endfunction
endclass

class frame_vc;
  agent agn;
  function new(...);
    agn = new(...);
  endfunction
endclass
```

Constructs next level *

- Aggregate class hierarchy
- Declared bottom-up
  - Cannot reference class before declaration
  - Compilation dependencies
- Create next level instances in constructor*
- Can have multiple instances at each level
- A driver in one of 2 agent instances has a problem
  - How do we know which driver?

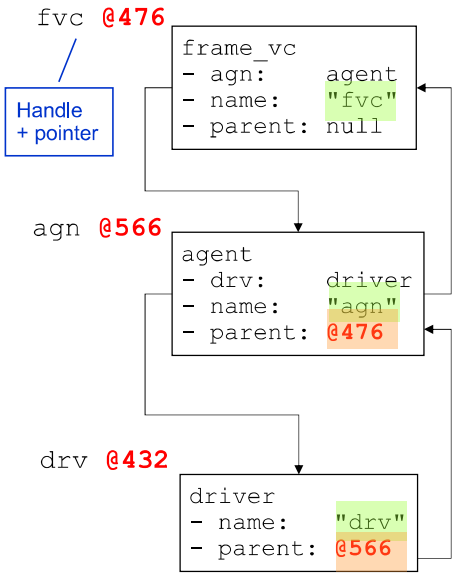
*In SystemVerilog. There are better options in UVM

65

The Verification Component is built as an aggregate class hierarchy. As we cannot reference a class before its declaration, we need to declare the classes bottom up. So the sequencer, driver and monitor classes are declared first. Then we can declare the agent class. The agent declares properties of the sequencer, driver and monitor classes and constructs instance for these properties in the agent constructor. The verification component class can then be declared and this creates an instance of the agent class in its constructor. So the Verification Component hierarchy is declared, and constructed, bottom-up.

As discussed previously, it is possible for a Verification Component to construct multiple instances of agent classes in an advanced application. If a driver in one of these instances has a problem, we need to which driver in which agent the problem occurred. The drivers may have the same instance name, but the agents should have different names (as they are instances at the same hierarchical level). Therefore to distinguish one driver from another, we need to know its position in the hierarchy, i.e., its pathname.

Instance Names and Parent Pointers



Two additional properties for every component:

- Instance name string to identify component
  - By convention, should match handle name
- Parent pointer
  - Link to the component which created current component
  - Parent of topmost component left unassigned (`null`)

A component can now generate its hierarchy path

1. Copy name to path
2. Jump to parent
3. Prepend name to path
4. Repeat steps 2-3 until parent = `null`

`fvc.agn.drv`

66

Component classes therefore have an extra property, in addition to the instance name we saw in data items.

The component still has an instance name string, which, by convention, we make equal to the handle name.

The component also has a parent pointer property. This is a pointer to the component which created the current instance. i.e. the driver will have a pointer to the agent which created it, and the agent will have a pointer to the VC which instantiated it.

So when the fvc component is created in memory with the pointer @476, its constructor creates an instance of the agent class in the pointer @566. The parent of the agent is set to the pointer of the fvc instance, @476. Likewise when the agent creates the driver instance, the parent of the driver is set to the pointer of the agent agn, @566. At the top of the component hierarchy, fvc in this case, the parent pointer is assigned to null as a terminating value for traversing the hierarchy.

Now, any component can generate its own hierarchical pathname using the parent pointers and instance names.

From driver, we copy the driver instance name (drv) to the path and follow the driver parent pointer to the @566 agent instance. We prepend the instance name of the agent (agn) to the path and follow the agent parent pointer to the @476 fvc instance. We prepend the fvc instance name to the path. The parent of fvc is null, indicating we have reached the top of the hierarchy, and our driver pathname, fvc.agn.drv is complete.

Implementation

```
class component_base;
  string name;
  component_base parent;

  function new (string name, component_base parent);
    this.name = name;
    this.parent = parent;
  endfunction

  function string pathname();
    ...
  endfunction : pathname

endclass

class uvc extends component_base;
 agent agn;

  function new (string name, component_base parent);
    super.new(name, parent);
    agn = new("agn", this);
  endfunction
...
```

Instance names and parent pointers common to all component classes

- Declare a component base class
  - String instance name
  - Parent pointer of base class type
  - Parent and name assigned in constructor

Extend all components from base

- Parent set to `this` in constructor call

Pathname implemented in base component

```
function string pathname();
   component_base ptr = this;
   pathname = name;
   while (ptr.parent != null) begin
     ptr = ptr.parent;
     pathname = {ptr.name, ".", pathname};
   end
 endfunction : pathname
```

Current name

Move to parent

Prepend name

67

The instance name and parent properties will be common to all component classes. Likewise hierarchy methods such as pathname. The most efficient implementation is to declare a component base class which contains members common to all components, and then extend every component from this base.

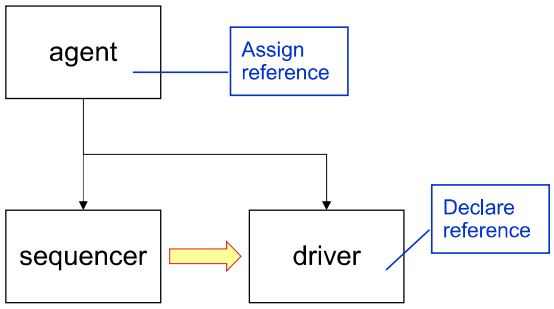
This also has the advantage that the parent pointer property can be declared of the component base type.

Both instance name and parent pointer are added as constructor arguments.

Every component class extends from component_base and has a constructor with the same name and parent arguments. These are passed up to component_base in a first line super call. Then the component can construct subcomponent instances. When constructing subcomponents, the name argument is set to a string matching the handle name (by convention) and the parent pointer is set via the keyword this.

The component_base class can also implement common methods such as pathname(). The slide shows a possible implementation of pathname using the algorithm from the previous slide. We use a local ptr declaration of type component_base to navigate the hierarchy. We construct the pathname in the function name, initialized to the current components instance name. While the parent pointer is not null (indicating the top of the hierarchy) we jump up one level by following the current components parent pointer, and then prepend the name of the new component to the pathname. using "." as a separator.

Simple Component Connections



Need to connect components

- Assign property of another component
- Access method of another component

Simple reference is easiest

- Declare handle on target component
- Assign from higher level

```
class sequencer extends component_base;
  ...
  function frame get_next_item();
  ...
endclass

class driver extends component_base;
  sequencer sref;
  ...
  data = sref.get_next_item();
  ...
endclass

class agent extends component_base;
  driver drv;
  sequencer seqr;

  function new(...);
    drv = new("drv",this);
    seqr = new("seqr",this);
    drv.sref = seqr;
  endfunction
endclass
```

Declaration order important

Reference handle – not constructed

Access method

Assign reference

68

Components may need to communicate with each other, for example to pass data items from the sequencer to the driver. Therefore we need some form of connection between components.

The simplest connection is by using a reference. We declare the sequencer class first. Then when we declare the driver, we add a property of the sequencer class. This is not a subcomponent. We will not construct an instance in this handle.

In a higher hierarchical level, usually the next level up which in this case is the agent, we assign the driver sequencer handle to the sequencer instance. Obviously we have to do this after we have constructed both driver and sequencer. The driver can now use it's sequencer handle to access anything from the sequencer instance.

This is a simple reference connection. There are better alternatives for sequencer/driver connection in UVM, but reference connections are still used in UVM.

Components Quiz

1. What is the issue with creating a component where the instance name does not match the handle name? For example:

```
class uvc extends component_base;
 agent agn;

  function new (string name, component_base parent);
    super.new(name, parent);
    agn = new("agent", this);
...
```

Handle name

Instance name

Solutions at end of module

69

This page does not contain notes.

Labs

Lab 5: Building a Component Hierarchy

- Extending component classes from a component base
- Creating a Verification Component hierarchy
- Simple reference connection
- Implementing run tasks to print the hierarchy

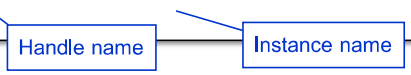
70

This page does not contain notes.

Components Quiz Solution

1. What is the issue with creating a component where the instance name does not match the handle name? For example:

```
class uvc extends component_base;
 agent agn;

  function new (string name, component_base parent);
    super.new(name, parent);
    agn = new("agent", this);
...
```



The issue is that a class-based verification environment will typically use instance name paths for reports, for example in the output of a `pathname()` method, and handle names paths in, for example, the hierarchical assignment of component properties. If the instance and handle paths match, an assignment path can be read from a `pathname()` output. If they differ, then you would have to open up each level to find the actual handle name path.

71

This page does not contain notes.

DUT Connection

Module	**8**
Revision	3.0
Version	1.2.5

This page does not contain notes.

Module Objectives

In this module, you will

- Connect classes to DUT RTL ports using SystemVerilog virtual interfaces

73

This page does not contain notes.

SystemVerilog Interface Review

frame_if.sv

```
interface frame_if (input logic clock);
  logic suspend, valid;
  logic [7:0] data;
...
```

run.f

```
...
-incdir ../sv
../sv/yapp_pkg.sv
../sv/frame_if.sv
...
```

Compile interface in `run.f`

An interface is a named bundle of nets or variables

Similar to a module

- Interfaces are instantiated inside modules
- Interface signals connected to module ports
  - Interface signals referenced via the "dot" operator
    - `interface_instance.signal_name`
- An interface can contain parameters, variables, functions, tasks, assertions and clocking blocks
- An interface must be compiled separately like a module (e.g. in `run.f`)
  - Cannot `` `include `` inside a package or other module

74

Remember an interface file is a design element like a module. It must be compiled separately by, e.g. adding to the list of compilation files in a run.f file. You cannot `include an interface into a package, or declare it inside a package directly.

Driving Data into DUT from Module

```
interface frame_if (input logic clock);
  logic suspend, valid;
  logic [7:0] data;
...
```

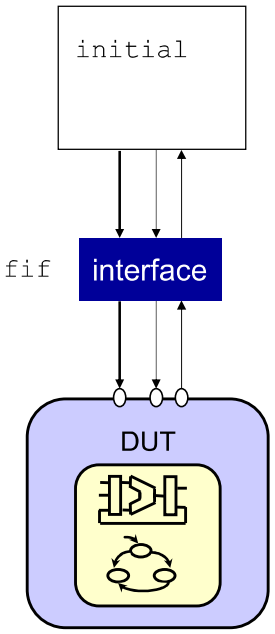
Interface declaration

```
module top;
  bit clock;
  frame frm;

  frame_if fif(clock);

  initial begin
    // create frame instance and randomize
    @(negedge fif.clock iff fif.suspend == 0);
    fif.valid = 1'b1;
    fif.data = {frm.addr, frm.length};
  ...

  frame_dut dut(
    .clock(clock),
    .suspend(fif.suspend),
    .valid(fif.valid),
    .data(fif.data),
    ...
```

Instantiate interface

Drive interface instance signals

Instantiate DUT and connect to interface instance signals



75

In module-based verification, we can drive the DUT via an interface as follows:

- Instantiate the interface with a suitable name, here fif.
- Drive the signals of the interface from an initial block of a module using the interface instance name as a prefix.
- In the port mapping of the DUT, connect the interface signals to the correct DUT ports, again using the instance name as a prefix.

Driving Data into DUT from Class Via Interface Instance

```
interface frame_if (input logic clock);
  logic suspend, valid;
  logic [7:0] data;
...
```

Interface declaration

```
module top;
  bit clock;

  frame_if fif(clock);
...
  class driver extends component_base;
  ...
    task send_to_dut(frame frm);
      @(negedge fif.clock iff fif.suspend == 0);
      fif.valid = 1'b1;
      fif.data = {frm.addr, frm.length};
    ...

  frame_dut dut(
    .clock(clock),
    .suspend(fif.suspend),
    .valid(fif.valid),
    .data(fif.data),
    ...
```

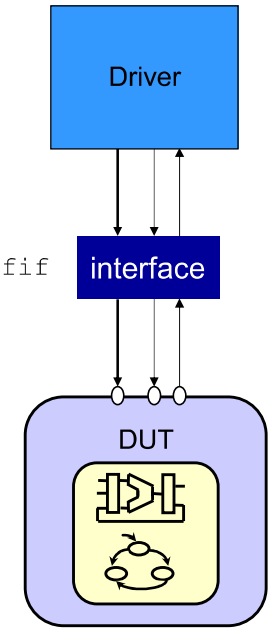
Instantiate interface

Drive interface instance signals

Instantiate DUT and connect to interface

Warning

This breaks reuse. Why?



76

In class-based verification, we could drive the DUT via an interface in the same way as for a module:

Instantiate the interface with a suitable name, here fif.

Drive the signals of the interface from the send_to_dut() task of the driver using the interface instance name as a prefix.

In the port mapping of the DUT, connect the interface signals to the correct DUT ports, again using the instance name as a prefix.

However this breaks reuse of the driver class and causes issues for compilation. Why?

Reuse Issues in Class Connection

```
interface frame_if (input logic clock);
  logic suspend, valid;
  logic [7:0] data;
...
```

```
module top;
  bit clock;

  frame_if fif(clock);
...
  class driver extends component_base;
  ...
    task send_to_dut(frame frm);
      @(negedge fif.clock iff fif.suspend == 0);
      fif.valid = 1'b1;
      fif.data = {frm.addr, frm.length};
    ...

  frame_dut dut(
    .clock(clock),
    .suspend(fif.suspend),
    .valid(fif.valid),
    .data(fif.data),
```

Declaration dependency

Hardwired to instance `fif`

Classes should not be directly connected to an interface instance

- Breaks reusability
  - If `driver` is hardwired to `fif`, cannot drive any other instance of `frame_if` such as `fif2`
- Interface instances are static
  - `driver` has to be declared local to the `fif` instance

What we need is an interface variable:

- Used in class to access interface signals
- Assigned to a specific interface instance when class instantiated

77

We don't want to reference a specific interface instance name (e.g. fif) in the verification component class.

Firstly, this breaks reusability. If we hardwire the class to drive interface instance fif, then the user must always use that instance name for their interface. This prevents the class from being used with a different instance name. Also it would prevent the class being used with a design which had multiple instances of a particular interface. For example, our 4-port switch would require 4 separate classes, each hardwired to drive a specific interface instance.

Secondly, interface instances are static. Therefore, a class hardwired to a specific interface instance has to be declared in the same scope as the interface instance. We could not declare the class in a more convenient scope for reuse such as a package.

SystemVerilog gives us a solution to this problem. We can use a virtual interface. This is a variable of a specific interface type. We declare the variable as a class property and access the interface signals via the variable. The variable can then be assigned to any instance of the interface type, allowing the user to use any interface instance name and have multiple instantiations of the class connected to different interface instances.

Virtual Interface

```
interface frame_if (input logic clock);
  logic suspend, valid;
  logic [7:0] data;
...
```

```
class driver extends component_base;

   virtual interface frame_if vif;

   function new (...
                 virtual interface frame_if vif);
      ...
      this.vif = vif;
   endfunction

  ...
    task send_to_dut(frame frm);
      @(negedge vif.clock iff vif.suspend == 0);
      vif.valid = 1'b1;
      vif.data = {frm.addr, frm.length};
    ...
```

Virtual interface

Set virtual interface in constructor

Virtual interface access to signals

A virtual interface is a variable which can be connected to an interface instance

Can be declared as a class property

- `interface` keyword is optional
  - Include for readability

Access interface signals using variable name as a prefix

Default value is null

- Must be assigned to an interface instance:
  - In the constructor
  - Via a method
  - By direct assignment

78

A virtual interface can be declared as a class property. The declaration uses the keywords virtual and interface. Then the interface type name and finally the name of the virtual interface. Since the user will need to know the name of the virtual interface in order to connect the class to the DUT, by convention the name is kept short and simple, like vif.

Also the interface keyword in the virtual interface declaration is optional. However the virtual keyword is used in classes for several different constructs (virtual classes, virtual methods) therefore for clarity, it is recommended to include this part of the declaration.

The virtual interface is then used as a prefix to access the signals of the interface.

The default value of a virtual interface is null. You must assign the virtual interface to an actual instance before using it. The virtual interface can be assigned in a constructor, as here, a separate configuration method or by direct assignment. For constructor or method assignment, you can declare a virtual interface function argument using the same syntax as for the declaration of the virtual interface property. Again the interface keyword is optional.

Interface Methods

```
interface frame_if (input logic clock);
...
  logic suspend, valid;
  logic [7:0] data;

  task send_to_dut(frame frm);
    @(negedge clock iff suspend == 0);
    valid = 1'b1;
    data = {frm.addr, frm.length};
    foreach( frm.payload [i]) begin
      @(negedge clock iff suspend == 0);
      data = frm.payload[i];
    end
  ...
  endtask: send_to_dut
...
```

```
class driver extends component_base;

  virtual interface frame_if vif;
  ...
  // get frame instance from sequencer
  vif.send_to_dut(frame frm);
...
```

Stimulus tasks can be declared in interfaces

- Called interface methods

Advantages

- Much better acceleration performance

Disadvantages

- Cannot use inheritance to modify protocol

TB: Future-Proof Your UVM Environments With Acceleration Optimization

79

Methods which access interface signals, such as the send_to_dut() task, can either be declared in a class, such as the driver, or declared in the interface. A task declared in the interface is called an interface method. An interface method is called from within a class using the virtual interface prefix, just like an interface signal.

Interface methods are recommended as they give significantly better acceleration performance. Interface methods are essential if your Verification Component is used in a hardware or software acceleration environment, or, crucially, will be used with acceleration at some point in the future.

Hence you can future-proof your verification components by using interface methods.

One disadvantage is you can no longer easily customize the send_to_dut() method, for example, by declaring a subclass of the existing driver and redefining the send_to_dut() task.

For more explanation on why interface methods are more efficient in a acceleration environment, please search for the following presentation on support.cadence.com.

Future-Proof Your UVM Environments With Acceleration Optimization

DUT Connection Quiz

1. What is the default value of a virtual interface?

2. In which ways can you assign a virtual interface class property? Can you suggest any advantages or disadvantages of each option?

Solutions at end of module

80

This page does not contain notes.

DUT Connection Quiz Solutions

1. What is the default value of a virtual interface?
   - `null`
2. In which ways can you assign a virtual interface class property? Can you suggest any advantages or disadvantages of each option?
   - In the constructor using a constructor argument (strictly speaking a subset of assigning via a method)
     - To pass a virtual interface from the VC to a driver or monitor via a constructor argument would require adding the virtual interface argument to constructors in both VC and agent. So the constructor arguments cascade down the VC hierarchy. This might be easy for a single virtual interface argument, but multiple component properties will quickly make the constructor argument lists too bloated to use. Therefore, generally, constructor arguments are a bad choice for passing component properties.
   - In a method using a subprogram argument
     - This is a better option as a separate configuration method can be written once in the agent or Verification Component and can directly assign the monitor/driver virtual interfaces via a pathname.
     - The user only needs to know the method name, not which components use interfaces, their internal pathnames or the names of the virtual interfaces.
   - Direct assignment
     - The user would need to know the internal pathnames of the VC, which VC components contain virtual interfaces and the virtual interface names. With a developer-written configuration method, this information is abstracted away from the user.

This page does not contain notes.

Building a Verification Component

Module **9**

Revision 3.0
Version 1.2.5

Module Objectives

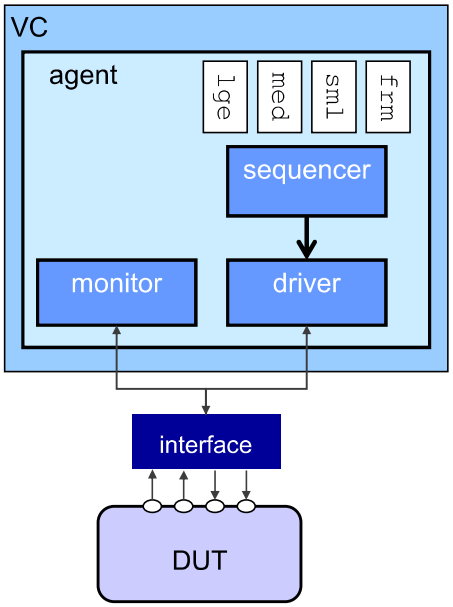
In this module, you will

- Use the knowledge gained so far to build a simple Verification Component (VC)
- Highlight common implementation issues for VCs

83

This page does not contain notes.

Verification Component Functionality



What must each component do?

- Sequencer
  - Generate and randomize data item
- Driver
  - Convert data item into signal transitions
  - Drive interface signals with correct protocol
- Monitor
  - Capture signal transitions on interface and regenerate data
- Agent
  - Instantiate sequencer/driver/monitor
- Verification component (VC) top level
  - Instantiate agent
  - Configure VC
    - Connect interfaces
    - Assign properties

84

Here is a reminder of the functionality of each component in a Verification Component hierarchy.

Sequencer

```
class sequencer extends component_base;
  bit ok;
  // Properties for stimulus generation

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

  function void get_next_item(output frame frm);
    frame  frm;
    smallframe small;
    randcase
      1: frm = new("frm");
      1: begin
         small = new("small");
         frm = small;
         end
      ...
    endcase
    ok = frm.randomize();
  endfunction
...
```

Constructor

Method to generate stimulus

Instantiation of `frame` or `smallframe`

Randomize instance

- `get_next_item` method creates frame instances
  - Randomly generates `frame` or `smallframe` instance
  - Sets frame data using assignment and/or `randomize`
  - Returns instance by `frame` handle argument
- Sequencer may have other properties for stimulus generation
  - For example, port number property to define `source` field of switch packet
  - Must be set in configuration

85

Here is the sequencer class for the Verification Component.

The method get_next_item() randomly creates either a frame or smallframe instance. A randcase statement is used to select which class instance to create. randcase weights can be used to obtain the right distribution and additional branches of the randcase can be used to randomly create other instances of frame subclasses. The method returns the frame instance via an argument of type frame so that it can hold instances of frame, smallframe or any subclass of frame. The instance is randomized before being returned.

The sequencer may contain other properties which are necessary for stimulus generation. For example, in the four-port switch lab design, the sequencer needs to know to which port it is connected in order to set the source field of the packet data. We would need a local property for the source information which would be set when the VC is configured.

Driver

```
class driver extends component_base;

  virtual interface uvc_if vif;
  sequencer sref;

  frame frm;

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

  task run (int runs);
    repeat (runs) begin
      sref.get_next_item(frm);
      vif.send_to_dut(frm);
    end
  endtask

endclass
```

Virtual interface

Sequencer reference

Constructor

Run task for functionality

Driver has:

- Constructor
- A virtual interface
  - To call interface stimulus method
  - Or access interface signals
- A pointer to the sequencer
  - To call sequencer method for next frame
- Run task
  - Argument defines number of frames to send
  - Gets next item...
  - ...sends to DUT

86

Here is the driver class for the Verification Component. It contains:

A constructor.

The virtual interface vif, which is used to access interface methods or interface signals.

A reference handle connection to the sequencer class. Remember we do not construct this handle sref. It is assigned in a higher level to the sequencer instance to which we connect the driver. The reference allows us to call methods declared in the sequencer.

The run task implementing the functionality of the driver. The task calls the sequencer get_next_item() method using the sequencer reference handle and assigns the output frame instance into the local handle frm. The task then calls the interface method send_to_dut() passing frm as the input argument, to drive the frame on the DUT ports. In this simple driver, we wrap the sequencer and interface method calls in a loop controlled by the run task runs argument to control the number of frames sent to the DUT.

Monitor

```
class monitor extends component_base;

 virtual interface frame_if vif;

 frame frm;

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

  task run();
   forever begin
     vif.collect_packet(frm);
     frm.print();
    end
  endtask

endclass
```

Virtual interface

Constructor

Run task for functionality

Forever collect and process frame

Monitor has:

- Constructor
- A virtual interface
  - To call interface stimulus method
  - Or access interface signals
- Run task
  - Calls interface method to collect frame from interface signals...
  - ... or reads interface signal directly
  - Processes result
    - Print for debug
    - Send to scoreboard
    - Apply coverage
  - Wrap collection and processing in `forever` loop

87

Here is the monitor class for the Verification Component. It contains:

A constructor.

The virtual interface vif, which is used to access interface methods or interface signals.

The run task implementing the functionality of the monitor. The task calls the interface method collect_packet(), with the frm frame handle as the output argument, to collect frame data from the DUT ports. Once collected, the monitor can process the collected data. Here we simply print the collected frame, but the monitor could also collect coverage information and pass data to a scoreboard. In this simple monitor, we wrap the interface method and print calls in a forever loop to constantly monitor the DUT ports.

Agent

```
class agent extends component_base;

 driver dd;
 sequencer sg;
 monitor mon;

  function new (string name, component_base parent);
    super.new(name, parent);
    dd = new("dd", this);
    sg = new("sg", this);
    mon = new("mon", this);

    dd.sref = sg;
  endfunction

endclass
```

Construct subcomponents

Connect driver and sequencer

Declares sequencer, driver and monitor subcomponent handles

Constructs subcomponent instances in its constructor

Driver's `sequencer` reference assigned to link driver and sequencer

- After both components have been constructed

A VC may have several different agents

- Transmit and receive
- Master, slave and arbitration

88

Instances of the sequencer, driver and monitor are constructed in the agent class. We use instance names and parent pointers in the component constructors as defined previously. The agent constructor must also assign the sref sequencer reference handle of driver to the sequencer instance sg. So when the driver calls the sequencer method get_next_item() from the sref reference, the correct instance is used for the call.

Remember a Verification Component may have several different agents types declared. For example, the VC may instantiate a transmit or receive agent depending on which end of a bus it is driving. A bus protocol may support several different types of connection, such as master, slave or arbitration. A complex VC may have agents for each connection type to allow the modelling of complex bus topologies. A VC may also support multiple instantiations of the same agent type, for example to send a stream of data over an array of DUT ports.

Verification Component

```
class frame_vc extends component_base;

  agent a1;

  function new (string name, component_base parent);
    super.new(name, parent);
    a1 = new("a1", this);
  endfunction

  function void configure(virtual interface frame_if vif);
    a1.drv.vif = vif;
    a1.mon.vif = vif;
  endfunction

  task run(int runs);
    fork
      agn.mon.run();
    join_none
    agn.drv.run(runs);
  endtask
endclass
```

Construct agent

Configuration

Initiate subcomponent run tasks

Agent instance created in verification component constructor

Have we finished?

- Need configuration
  - Driver and monitor interfaces to be connected
  - Any component properties to be assigned
- Need to initiate component functionality
  - Call driver run task
  - Spawn monitor run in concurrent process
    - As it contains a `forever` loop

89

Here is the Verification Component class. It contains handles on one or more agent class and constructs the agent instance(s) in its constructor.

The VC also needs to be configured. At the minimum we need to assign the virtual interfaces of the driver and monitor components. The interface instance is passed as an argument to configure and assigned to the driver and monitor virtual interfaces via hierarchical pathnames. Configure could also assign other properties such as the port number for our four-port switch lab example.

Finally we need to initiate the functionality of the VC. The VC has a run task which calls the driver run task with the appropriate argument, and also executes the monitor run task. Remember the monitor run task contains a forever loop. We cannot just call that task directly. It may cause the simulation to hang. For safety, we use a SystemVerilog fork-join_none to execute the monitor run task in a separate, concurrent simulation process.

Instantiate Verification Component

```
module framevc_test;
  import frame_pkg::*;

  logic clk = 1'b0;
  logic reset = 1'b0;;

  frame_vc frm1;

  frame_if f0 (clk, reset);

  always
    #10 clk <= ~clk;

  initial begin
    // reset generation
    frm1 = new("frm1", null);
    frm1.configure(f0);
    frm1.run(3);
    ...
    $finish;
  end
...
```

Import package

VC handle

Interface instance

Clock generator

VC construct, configure and run

Top level module:

- Imports Verification Component (VC) package
- Generates clock and reset signals etc.
- Instantiates DUT
- Instantiates interface and connects to DUT
- Constructs VC
- Configures VC
- Executes VC run

In a realistic environment, there will be other layers

- See UVM training...

90

Finally we have a topmost module to declare all the components of the testbench environment. This topmost module must:

Import the package containing the Verification Component declarations.

Generate clock and reset signals.

Instantiate the top module of the DUT.

Instantiate interfaces and connect these to the DUT ports.

Construct an instance of the verification component with the parent argument null.

Configure the verification component instance with the interface instance.

Execute the run method of the verification component to generate a set number of randomly selected packets.

Building a Verification Component Quiz

1. Can you identify any issues with the simple Verification Component described this module?

Solutions at end of module

91

This page does not contain notes.

Lab

Lab 6: Completing the Verification Component

- Generating stimulus in sequencer
- Adding virtual interfaces to driver and monitor
- Getting stimulus and driving DUT in driver
- Capturing DUT data in monitor
- Initial standalone testing
- Testing with DUT

92

This page does not contain notes.

Building a Verification Component Quiz Solution

1. Can you identify any issues with the simple Verification Component described this module?

There are many issues, but here are a few:

1. The sequencer just creates random packets. We need better control over stimulus to allow us to specify the exact number, type and properties of the packets generated in the sequencer.
2. Having to specify the number of runs is limiting. A more flexible solution would, for example, automatically execute the number of packets requested in the sequencer.
3. Automatically triggered run tasks would be better than manually triggered.
4. Connecting components using references is not very user-friendly or reusable. It might be fine for internal VC connections, but for connecting a monitor to a scoreboard, we would want a better approach.
5. Configure methods are executed after component construction, and therefore cannot assign properties to control construction. For example, the number of agents constructed.
6. Printing packets requires user-defined code. It would be better if printing, copying and comparing was automatically generated.

This page does not contain notes.