ASIC Verification

Advanced Object Oriented Programming

Fall 2011
Meeta Yadav
Topics

• Inheritance
• Polymorphism
• Abstract Classes
• Parameterized Classes
Object Oriented Programming (OOP): Introduction

• Why OOP?
  ◆ Helps in creating and maintaining large testbenches:
    ▶ You can create complex data types and tie them together with the routines that
      work with them
  ◆ Increases productivity:
    ▶ You can create testbenches and system-level models at a more abstract level by
      calling a routine to perform an action rather than toggling bits
    ▶ You can work with transactions rather than signal transitions
  ◆ Allows the Testbench to be reused:
    ▶ OOP decouples the testbench from design details making it more robust and
      easier to maintain and reuse

• How should the data and code be brought together
  ◆ The data that flows in and out of the design is grouped together in
    transactions so the easiest way to organize the testbench is around the
    transactions and the operations that you perform
OOP Basics: Terminology

Blueprint for a house | A complete house | House Address
123 Elm Street

Class | Object | Handle

Turn on/off switches | Light switches

Methods | Properties
OOP Basics: Terminology

- **Class**
  - Programming element “containing” related group of features and functionality
  - Encapsulates functionality
  - Provides a template for building objects
  - Can be used as data structures

- **Object**
  - An object is an instance of a class

- **Handle**
  - A type safe pointer to the object and cannot be modified

- **Properties**
  - Variables contained in the instance of the class

- **Methods**
  - Tasks/functions (algorithms) that operate on the properties in this instance of the class
Let's create a layered testbench
Simplified Layered Testbench
Creating Transactions

• How should I model my transactions?

Create a simple transaction class to create transactions

Class Transaction;
  rand bit [31:0] src, dst, data[8];
  bit [31:0] crc;

  function void calc_crc;
    crc = src ^ dst ^ data.xor;
  endfunction

  function void display(input string prefix="");
    $display("%sTr: src=%h, dst=%h, crc=%h",prefix, src, dst, crc);
  endfunction
endclass

Class creates a simple class that sends a transaction into the design
Creating a Layered Testbench

- **Driver**

```verilog
Class Driver;
    mailbox gen2drv;

    function new(input mailbox gen2drv);
        this.gen2drv = gen2drv;
    endfunction

task main;
    Transaction tr;    // Handle to a Transaction object or
    // a class derived from Transaction

    forever begin
        gen2drv.get(tr);   // Get transaction from generator
        tr.calc_crc();    // Process the transaction
        @ifc.cb.src = tr.src;   // Send transaction
        ...
    end
endtask
endclass
```
Creating a Layered Testbench

• **Generator**

```verbatim
Class Generator;
    mailbox gen2drv;
    Transaction tr;

    function new(input mailbox gen2drv);
        this.gen2drv = gen2drv;  // this-> class-level var
    endfunction

task run();
    forever begin
        tr = new();  // Construct transaction
        assert(tr.randomize());  // Randomize it
        gen2drv.put(tr);  // Send to driver
    end
endtask
endclass
```

This generator sends in an transaction with default constraints and does not offer a way to change it.
Creating a Layered Testbench

Environment

```plaintext
Class Environment;
    Generator gen;
    Driver drv;
    mailbox gen2drv;

    function void build(); // Build the environment
        gen2drv = new();  // constructing the mailbox,
        gen = new(gen2drv); // generator,
        drv = new(gen2drv); // and driver
    endfunction

    task run();
        fork
            gen.run();
            drv.run();
        join
    endtask

    task wrap_up();
        // Empty for now - call scoreboard for report
    endtask
endclass
```
Creating a Layered Testbench

Layered Testbench

Program automatic test;

    Environment env;
    initial begin
        env = new();            // Construct the environment
        env.build();           // Build testbench objects
        env.run();            // Run the test
        env.wrap_up;          // Clean up afterwards
    end
Endprogram
How do I change my Transaction class?
Creating Complex Transactions

• How should I inject errors and variable delays into my design?

Create a large flat transaction class that can
1. Create a normal transaction
2. Inject errors into the design

Class Transaction;
    rand bit [31:0] src, dst, data[8];
    bit [31:0] crc;
    function void calc_crc;
        crc = src ^ dst ^ data.xor;
    endfunction

    function void display(...);
        ...
    endfunction

    function bad_crc(input bit bad);
        if(bad) crc=~crc;
    endfunction
endclass

Class creates a large class that sends a transaction into the design and injects errors as well
Creating Complex Transactions

• Problems with a large flat class
  ◆ Slow to develop and debug
  ◆ Results in a large class that is a burden to maintain
  ◆ Any changes in the transaction will require changes to the class structure
Creating Complex Transactions

- How should I model my transactions now?

**Solution 2:** Can we somehow extend the class and add the additional functionality?

<table>
<thead>
<tr>
<th>Base Class</th>
<th>Extended Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>src</td>
<td>bad_crc</td>
</tr>
<tr>
<td>dst</td>
<td>calc_crc</td>
</tr>
<tr>
<td>data[8]</td>
<td></td>
</tr>
<tr>
<td>crc</td>
<td>super.calc_crc</td>
</tr>
<tr>
<td>display</td>
<td>BadTr</td>
</tr>
<tr>
<td></td>
<td>BadTr=Transaction+bad_crc</td>
</tr>
</tbody>
</table>

Need to access this calc_crc
BadTr can see all the variables in the base class
BadTr can see all the variables in the base class
BadTr can see all the variables in the base class
Creating Complex Transactions

- How should I model my transactions now?

```
Class Transaction;
  rand bit [31:0] src, dst, data[8];
  bit [31:0] crc;

  function void calc_crc;
    crc = src ^ dst ^ data.xor;
  endfunction

  function void display(...);
    ...
  endfunction
endclass

Base Class

Class BadTr extends Transaction;
  rand bit bad_crc;

  function void calc_crc;
    super.calc_crc();
    if(bad_crc) crc=~crc;
  endfunction

endclass

Extended Class
```

This Property is called Inheritance
Inheritance Summary

• How do I share code between classes?
  ◆ Instantiate a class within another class
  ◆ Inherit from one class to another (inheritance/derivation)

• Inheritance allows you to ‘add’ extra:
  ◆ Add extra Properties (data members)
  ◆ Add extra Methods
  ◆ Change the behavior of a method

• Common code can be grouped into a base class
  ◆ Additions and changes can go into the derived class

• Advantages:
  ◆ Reuse existing classes from previous projects with less debug
  ◆ Won’t break what already works
Inheritance Summary

• Hierarchy
  ◆ Can only go up one level
    ▶ super.variable_name
  ◆ Cannot go across multiple levels
    ▶ super.super.variable_name is illegal
Inheritance: Redefining Routines

- How do I redefine the routines in the extended class?

Declare routines as virtual
Inheritance: Redefining Routines

- **Virtual Routines**
  - Always defined routines in a class as virtual so that they can be redefined in the extended class

```markdown
Class Transaction;
    rand bit [31:0] src, dst, data[8]; // Random variables
    bit [31:0] crc;

    virtual function void calc_crc;
        crc = src ^ dst ^ data.xor;
    endfunction

    virtual function void display(input string prefix="");
        $display("^sTr: src=%h, dst=%h, crc=%h", prefix, src, dst, crc);
    endfunction
endclass
```

---

**Base Class**
Inheritance: Redefining Routines

- **Virtual Routines**

```vhdl
Class BadTr extends Transaction;
    rand bit bad_crc;

    virtual function void calc_crc;
        super.calc_crc(); // Compute good CRC
        if (bad_crc) crc = ~crc; // Corrupt the CRC bits
    endfunction

    virtual function void display(input string prefix="");
        $write("%sBadTr: bad_crc=%b, ", prefix, bad_crc);
        super.display();
    endfunction
Endclass : BadTr
```

**Extended Class**
Virtual Methods

Calling class methods

```
Transaction tr;
BadTr bad;

initial begin
  tr = new();
  tr.calc_crc(); // Calls Transaction::calc_crc

  bad = new();
  bad.calc_crc(); // Calls BadTr:::calc_crc

  tr = bad; // Base handle points to ext obj
  tr.calc_crc();  
end
```

If we left out `virtual` modifier on `calc_crc`, SV would call `Transaction::calc_crc`

The OOP term for multiple routines sharing the common name is polymorphism
Blueprint Patterns

Blueprint Pattern Generator

Blueprint Pattern Generator With New Pattern

Using Blueprint Patterns

Changing the generator for blueprint patterns

Class Generator;
    mailbox gen2drv;
    Transaction blueprint;

    function new(input mailbox gen2drv);
        this.gen2drv = gen2drv;
        blueprint = new();
    endfunction

    task run();
        Transaction tr;
        forever begin
            assert(blueprint.randomize);
            tr = blueprint;
            gen2drv.put(tr);        // Send to driver
        end
    endtask
endclass
Using Blueprint Patterns

Sending in the BadTr transaction

Program automatic test;

Environment env;
initial begin
  env = new();
  env.build(); // Construct generator, etc.
begin
  BadTr bad = new(); // Replace blueprint with
  env.gen.blueprint = bad; // the “bad” one
end

  env.run(); // Run the test with BadTr
  env.wrap_up(); // Clean up afterwards
end
endprogram
Changing Random Constraints

Changing random constraints using extended class

```
Class Nearby extends Transaction;
    constraint c_nearby {
        dst inside {[src-100:src+100]};
    }
Endclass

Program automatic test;
    Environment env;
    initial begin
        env = new();
        env.build(); // Construct generator, etc.
        begin
            Nearby nb = new(); // Create a new blueprint
            env.gen.blueprint = nb; // Replace the blueprint
            end
            end.run(); // Run the test with Nearby
            env.wrap_up(); // Clean up afterwards
        end
endprogram
```

Base_class = Extended_class

The other way is not allowed
Abstract Classes and Virtual Methods

• virtual class
  ◆ A set of classes can be created that can be viewed as being derived from a common based class
    ▶ For instance a common base class of the type BasePacket that sets out the structure of the packet is never instantiated but extended to derive useful subclasses
    ▶ Since the base class is not intended to be instantiated it can be made abstract by specifying it as virtual

```plaintext
virtual class BasePacket;
  virtual function integer send(bit[31:0] data);
  endfunction
endclass

class EtherPacket extends BasePacket;
  function integer send(bit[31:0] data);
    // body of the function
    ...
  endfunction
endclass
```
Parameterized Classes

- **Parameterized classes**
  - It is often useful to define a generic class whose objects can be instantiated to have different array sizes or data types
  - The normal verilog parameter mechanism is used to parameterize the class

```verilog
class vector #(int size = 1);
  bit [size-1:0] a;
endclass
```

- Instances of this class can then be instantiated like modules or interfaces

```verilog
vector #(10) vten; // object with vector of size 10
vector #(.size(2)) vtwo; // object with vector of size 2
typedef vector#(4) Vfour; // Class with vector of size 4
```
Thank You