ASIC Verification

Object Oriented Programming

Fall 2011
Meeta Yadav
Topics

• What is OOP
• Terminology
• An example class
• Default methods for classes
• Static attribute
• Assignment and copying
• Inheritance
• Polymorphism
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

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
## OOP Basics: Terminology

<table>
<thead>
<tr>
<th></th>
<th>HDL</th>
<th>OOP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Block definition</strong></td>
<td>module</td>
<td>class</td>
</tr>
<tr>
<td><strong>Block instance</strong></td>
<td>instance</td>
<td>object</td>
</tr>
<tr>
<td><strong>Block name</strong></td>
<td>instance name</td>
<td>handle</td>
</tr>
<tr>
<td><strong>Data Types</strong></td>
<td>registers &amp; wires</td>
<td>Properties: Variables</td>
</tr>
<tr>
<td><strong>Executable Code</strong></td>
<td>behavioral blocks (always, initial), tasks, functions</td>
<td>Methods: tasks and functions</td>
</tr>
<tr>
<td><strong>Communication between blocks</strong></td>
<td>Ports or cross-module task calls</td>
<td>calls, mailboxes, semaphores, etc.</td>
</tr>
</tbody>
</table>
OOP Basics: Advantages

• Traditional programming deals with data structures and algorithms separately
• OOP organizes transactions and transactors better through *encapsulation*
  ◆ Classes *encapsulate (group)* data and algorithms together logically
  ◆ Objects are just an instance of a class
  ◆ Classes are composed of "members".
  ◆ Members are either properties (data/variables) or methods (functions/tasks).
• OOP allows for characteristics & functionality of existing classes to be extended - *inheritance*
• OOP enables binding of data with functions at runtime - *polymorphism*
OOP: Your First Class

• Your first class
  ◆ A class encapsulates the data together with the routines that manipulate it
    ▶ A class’s data is referred to as class properties
    ▶ Subroutines of a class are called methods

```vhdl
class BusTran;

  bit [31:0] addr, crc, data[8];

  function calc_crc;
    crc=addr^data.xor;
  endfunction: calc_crc

  function display;
    $display("BusTran: %h", addr);
  endfunction: display

endclass: BusTran
```

Simple BusTran Class
Creating New Objects

- **Objects (Class Instance)**
  - An object is an instance of a class

  ```
  BusTran b;
  b = new();
  ```

  Declare a handle that points to an object of the type `BusTran`. When a handle is declared it is initialized to null.

  Call the `new` function to construct the `BusTran` object. When you call `new` you are allocating a new block of memory to store variables for that object.
  1. `new` allocates space for `BusTran`
  2. Initializes the variables to their default value (0 for 2 state and x for 4-state variables)
  3. Returns the address where the object is stored

*BusTran has two 32-bit registers (`addr` and `crc`) and an array with 8, 32 bit entries.*

*How much space would `new` allocate for an object of `BusTran`?*

\[
\text{Space} = 32 + 32 + 32 \times 8 = 320 \text{ bits of storage}
\]
Handles

• Getting a handle on objects
  ◆ Three basic steps to using a class are
    ▶ Defining class
    ▶ Declaring a handle
    ▶ Constructing an object

```
BusTran b1, b2;
b1 = new();
b2 = b1;
b1 = new();
```

Declare two handles
Allocate BusTran object
b1 & b2 point to it
Allocate second BusTran object
Object Deallocation

- Deallocating a handle on objects
  - SystemVerilog deallocates an object if there are no handles pointing to it
  - SystemVerilog does not deallocate an object that is being referenced by a handle
  - Need to manually clear all handles by setting them to null

```
BusTran b;
b = new();
b = new();
b = null;
```

- Declare a handle
- Allocate BusTran object
- Allocate another BusTran object and free the first
- Deallocate the second BusTran Object
Using Objects

- Refer to variables and routines using the . notation

```{}
class BusTran;
  bit [31:0] addr, crc, data[8];
  function calc_crc;
    crc=addr^data;
  endfunction: calc_crc
  function display;
    $display("BusTran: %h", addr);
  endfunction: display
endclass: BusTran
```

BusTran b;
b= new();
b.addr=32’h42;
b.display();
Initializing Class Properties

- Initialize the class properties when the object is constructed using inbuilt `new` function or user defined `new` function

  - **User defined new function allows the user to set the values as they prefer**

```systemverilog
class BusTran;
    logic [31:0] addr, src, data[8];
    function new();
        addr = 3;
        foreach(data[i])
            data[i] = 5;
    endfunction
endclass

class Driver;
    BusTran bt;
    function new();
        bt = new();
    endfunction
endclass: Driver
```

How does SystemVerilog know which new function to call?
- it does that by looking at the type of handle
Static Variables

- How do I create a variable shared by all objects of a class, but not make it global?
  - SystemVerilog allows you to create a static variable inside a class
    - The static variable is associated with the class definition, not the instantiated object
    - It is often used to store meta-data, such as number of instances constructed
    - It is shared by all objects of that class

```verbatim
class Transaction;
    static int count = 0;
    int id;
    function new();
        id = count++;
    endfunction
endclass
```

Using a id field can help keep track of transactions as they flow through test
Static Variables

Transaction b1,b2;
   initial begin
      b1=new; //first instance, id=0
      b2=new; //second instance, id=1
      $display(b2.id, b2.count)
   end

First instance id=0, count=1
Second instance id=1, count=2
1,2

• There is only one copy of the static variable count regardless of how many BusTran objects are created

• The variable id is not static so every BusTran has its own copy
Class Routines

- A routine in a class is a function or a task defined inside the scope of a class

```verilog
class BusTran;
    bit [31:0] addr, crc, data[8];
    function void display;
        $display("BusTran", addr, crc);
    endfunction: display
endclass: BusTran

class PCITran;
    bit [31:0] addr, data[8];
    function void display;
        $display("PCITran: %h", addr, data);
    endfunction: display
endclass: PCITran
```

```verilog
BusTran b;
PCITran pc;
initial begin
    b=new();
    b.display();
    pc=new();
    pc.display();
end
```
Using One Class Inside Another

- A class can contain an instance of another class, using a handle to an object
  - This is similar to Verilog’s concept of instantiation
  - Done for reuse and controlling complexity

```verilog
class BusTran;
    bit [31:0] addr, src, data[1024], crc;
    Statistics stats;
endclass

class Statistics;
    time startT, stopT;
    static int ntrans=0;
    static time total_elapsed_time;
endclass
```
Using One Class Inside Another

- Statistics class example

```plaintext
class Statistics;
    time startT, stopT;       //Transaction time
    static int ntrans=0;      //Transaction count
    static time total_elapsed_time=0;

    function time how_long;
       how_long=stopT-startT;
       ntrans++;
       total_elapsed_time +=how_long;
    endfunction

    function void start;
       startT=$time;
    endfunction
endclass
```
Using One Class Inside Another

• Using hierarchical syntax

```plaintext
class BusTran;
    bit [31:0] addr, src, data[1024], crc;
    Statistics stats;

    function new();
        stats = new();
    endfunction

    task create_packet();
        // Fill packet with data
        stats.start();
        // Transmit packet
    endtask
endclass
```

Use hierarchical syntax

Instantiate the object
Handles

• Getting a handle on objects
  ◆ Shallow copy

```
BusTran b1, b2;
b1 = new();
b2 = new b1;
```

Declare two handles
Allocate BusTran object
Shallow copy
Handles

- Example:

```plaintext
class A;
    integer j = 5;
endclass

class B;
    integer i = 1;
    A a = new;
endclass

function integer test;
    B b1 = new; // Create an object of class B
    B b2 = new b1; // Create an object that is a copy of b1
    b2.i = 10; // i is changed in b2, but not in b1
    b2.a.j = 50; // change a.j, shared by both b1 and b2
    test = b1.i; // test is set to 1 (b1.i has not changed)
    test = b1.a.j; // test is set to 50 (a.j has changed)
endfunction
```

---

©2011, Meeta Yadav, www.asicverification.org
Handles

• Getting a handle on objects
  ◆ Class properties and instantiated objects can be initialized directly in a class declaration
  ◆ Shallow copy does not copy objects
  ◆ Instance qualifications can be chained as needed to reach through objects
    \texttt{b1.a.j}
  ◆ Can write custom code to do a full copy of everything including nested objects

```java
BusTran b1,b2;
b1 = \texttt{new};
b2 = \texttt{new};
b2.copy(b1);
```
Deep copy
Handles

• Deep Copy

```verilog
class A;
    integer j = 5;
endclass

class B;
    integer i = 1;
    A a = new;
endclass

function integer test;
    B b1 = new; // Create an object of class B
    b1.a.j = 10; // j is changed in a
    B b2 = new; // Create an object of class B
    b2.copy(b1); // Perform deep copy
    b2.a.j = 50; // change b2.a.j,
    test = b1.a.j; // test is set to 10 (b1.a.j has not changed)
    test = b2.a.j; // test is set to 50 (b2.a.j has changed)
endfunction
```

```
j=5
j=10
j=10
j=50
```

```
B b1 = new;
b1.a.j = 10;

B b2 = new;
b2.copy(b1)
b2.a.j = 50;

```

```
test=10
test = b1.a.j;
test = b2.a.j;
```
• Write code for the sequence of handles and operations shown

```plaintext
class Thing;
    int data;
endclass

...  
Thing t1, t2;  // Two handles
initial begin
    t1 = new();  // Construct first thing
    t1.data = 1;
    t2 = new();  // Construct second
    t2.data = 2;
    t2 = t1;     // Second thing is lost
    t2.data = 5; // Modifies first thing
    $display(t1.data); // Displays "5"
end
```
Thank You
Inheritance

• 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

- Add additional functionality to an existing class

```
class Transaction;
    bit [31:0] src, dst, data[1024], crc;
endclass

class BadTr extends Transaction;
    bit bad_crc;
endclass

BadTr bt;
btt = new;
btt.src = 42;
btt.bad_crc = 1;
```
Inheritance

• Change the current functionality of a class: Single Inheritance
  - super keyword is used from within the extended class to refer to members of the parent class
  - It is necessary to use super to access members of a parent class when those members are overridden by the derived class

```plaintext
class Transaction;
  bit [31:0] src, dst, data[1024], crc;
  function void calc_crc();
    crc = src ^ dst ^ data.xor;
  endfunction
endclass

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

Want to access this method
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
```
Polymorphism

- **Dynamic method lookup**
  - Polymorphism allows the use of a variable in the superclass to hold subclass objects and to reference the methods of those subclasses directly from the superclass variable
    - For instance BasePacket defines, as virtual functions, all of the public methods that are to be used by the subclasses
  
    ```
    BasePacket packets[100];
    ```

  - Now instances of various packet objects can be created and put into the array
    ```
    EtherPacket ep = new; // extends BasePacket
    TokenPacket tp = new; // extends BasePacket
    GPSSPacket gp = new; // extends EtherPacket
    packets[0] = ep;
    packets[1] = tp;
    packets[2] = gp;
    ```

  - For instance `packets[1]` invokes send method associated with the TokenPacket class
    ```
    packets[1].send()
    ```
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
```