Chapter 9: Classes and Objects

Introduction

SystemVerilog introduces object-oriented programming (OOP) concepts to hardware description and verification. Classes provide a powerful way to create reusable, modular code structures that can model complex data types and behaviors. This chapter covers the fundamental concepts of classes and objects in SystemVerilog.

Class Declarations

A class in SystemVerilog is a user-defined data type that encapsulates data (properties) and functions (methods) that operate on that data. Classes serve as templates for creating objects.

Basic Class Syntax

```
class ClassName;
    // Properties (data members)
    // Methods (functions and tasks)
endclass
```

Simple Class Example

```
class Packet;
    // Properties
    bit [7:0] header;
    bit [31:0] payload;
    bit [7:0] checksum;

    // Method to display packet contents
    function void display();
        $display("Header: %h, Payload: %h, Checksum: %h",
                 header, payload, checksum);
    endfunction
endclass
```

Class with Constructor

```
class Transaction;
    rand bit [31:0] addr;
    rand bit [31:0] data;
    bit [1:0] cmd;

    // Constructor
    function new(bit [1:0] command = 0);
```

```
        cmd = command;
        // Randomize other fields
        assert(randomize());
    endfunction

    // Method to check transaction validity
    function bit is_valid();
        return (addr != 0 && cmd != 2'b11);
    endfunction
endclass
```

Properties and Methods

Properties are the data members of a class, while methods are the functions and tasks that operate on the class data.

Property Types

```
class DataPacket;
    // Basic properties
    bit [7:0] id;
    int length;
    real timestamp;
    string source;

    // Array properties
    bit [7:0] data[];
    int status_flags[4];

    // Random properties
    rand bit [15:0] sequence_num;
    randc bit [3:0] priority;

    // Constraints on random properties
    constraint valid_priority {
        priority inside {[1:8]};
    }

    constraint data_size {
        length > 0;
        length < 1024;
        data.size() == length;
    }
endclass
```

Method Types

```
class NetworkPacket;
    bit [47:0] src_mac;
    bit [47:0] dst_mac;
    bit [15:0] ethertype;
    bit [7:0] payload[];
```

```
    // Constructor
    function new(bit [47:0] src = 0, bit [47:0] dst = 0);
        src_mac = src;
        dst_mac = dst;
        ethertype = 16'h0800; // IP
    endfunction

    // Function method (returns a value)
    function int get_payload_size();
        return payload.size();
    endfunction

    // Task method (can consume time)
    task send_packet();
        #10ns; // Simulate transmission delay
        $display("Packet sent from %h to %h", src_mac, dst_mac);
    endtask

    // Virtual method (can be overridden)
    virtual function void print_header();
        $display("SRC: %h, DST: %h, Type: %h",
                 src_mac, dst_mac, ethertype);
    endfunction

    // Static method (belongs to class, not instance)
    static function bit [15:0] calculate_checksum(bit [7:0] data[]);
        bit [15:0] sum = 0;
        foreach(data[i]) sum += data[i];
        return ~sum;
    endfunction
endclass
```

Object Creation and Destruction

Objects are instances of classes created using the `new()` constructor. SystemVerilog handles memory management automatically.

Object Creation

```
class ConfigBlock;
    bit [31:0] base_addr;
    bit [7:0] version;
    bit enable;

    function new(bit [31:0] addr = 32'h1000);
        base_addr = addr;
        version = 8'h01;
        enable = 1'b1;
    endfunction

    function void configure(bit [31:0] addr, bit en);
        base_addr = addr;
        enable = en;
    endfunction
```

```
endclass

// Usage example
module test_objects;
    ConfigBlock cfg1, cfg2, cfg3;

    initial begin
        // Create objects
        cfg1 = new();                    // Use default constructor
        cfg2 = new(32'h2000);           // Pass parameter to constructor
        cfg3 = new(32'h3000);

        // Use objects
        cfg1.configure(32'h1500, 1'b0);
        cfg2.version = 8'h02;

        // Display object contents
        $display("Config 1: Addr=%h, Ver=%h, En=%b",
                 cfg1.base_addr, cfg1.version, cfg1.enable);
        $display("Config 2: Addr=%h, Ver=%h, En=%b",
                 cfg2.base_addr, cfg2.version, cfg2.enable);
    end
endmodule
```

Object Assignment and Copying

```
class DataBuffer;
    bit [7:0] buffer[];
    int size;

    function new(int sz = 16);
        size = sz;
        buffer = new[sz];
    endfunction

    // Deep copy method
    function DataBuffer copy();
        DataBuffer new_buf = new(size);
        new_buf.buffer = new[size];
        foreach(buffer[i]) new_buf.buffer[i] = buffer[i];
        return new_buf;
    endfunction

    function void fill_random();
        foreach(buffer[i]) buffer[i] = $random;
    endfunction
endclass

// Usage
module test_copy;
    DataBuffer buf1, buf2, buf3;

    initial begin
        buf1 = new(32);
```

```
        buf1.fill_random();

        buf2 = buf1;        // Shallow copy (both handles point to same object)
        buf3 = buf1.copy(); // Deep copy (creates new object)

        // Modify original
        buf1.buffer[0] = 8'hFF;

        // buf2 sees the change, buf3 doesn't
        $display("buf1[0] = %h", buf1.buffer[0]); // FF
        $display("buf2[0] = %h", buf2.buffer[0]); // FF (same object)
        $display("buf3[0] = %h", buf3.buffer[0]); // original value
    end
endmodule
```

The `this` Keyword

The `this` keyword refers to the current object instance and is used to resolve naming conflicts or for explicit reference.

```
class Counter;
    int count;
    string name;

    function new(string name, int count = 0);
        this.name = name;    // Distinguish parameter from property
        this.count = count;  // Explicit reference to object property
    endfunction

    function void increment(int count = 1);
        this.count += count; // Use this to access object property
    endfunction

    function Counter get_copy();
        Counter copy = new(this.name, this.count);
        return copy;
    endfunction

    function void compare_with(Counter other);
        if (this.count > other.count)
            $display("%s (%0d) > %s (%0d)",
                     this.name, this.count, other.name, other.count);
        else if (this.count < other.count)
            $display("%s (%0d) < %s (%0d)",
                     this.name, this.count, other.name, other.count);
        else
            $display("%s and %s have equal counts (%0d)",
                     this.name, other.name, this.count);
    endfunction
endclass

// Usage
module test_this;
    Counter c1, c2;
```

```
    initial begin
        c1 = new("Counter1", 5);
        c2 = new("Counter2", 3);

        c1.increment(2);
        c1.compare_with(c2);

        c2 = c1.get_copy();
        c2.name = "Counter2_copy";
        c1.compare_with(c2);
    end
endmodule
```

Class Scope and Lifetime

Class scope defines the visibility of class members, while lifetime determines when objects are created and destroyed.

Access Control

```
class SecureData;
    // Public members (default)
    string public_info;

    // Protected members (accessible in derived classes)
    protected bit [31:0] protected_key;

    // Local members (private to this class)
    local bit [127:0] private_data;
    local bit [7:0] secret_code;

    function new(string info = "default");
        public_info = info;
        protected_key = $random;
        private_data = {$random, $random, $random, $random};
        secret_code = 8'hA5;
    endfunction

    // Public method to access private data
    function bit [31:0] get_hash();
        return private_data[31:0] ^ protected_key ^ {24'b0, secret_code};
    endfunction

    // Protected method for derived classes
    protected function bit [31:0] get_protected_key();
        return protected_key;
    endfunction

    // Local method (private)
    local function bit verify_secret(bit [7:0] code);
        return (code == secret_code);
    endfunction
endclass
```

```
// Extended class demonstrating scope
class ExtendedSecureData extends SecureData;
    function new(string info = "extended");
        super.new(info);
    endfunction

    function void show_protected();
        // Can access protected members
        $display("Protected key: %h", protected_key);
        $display("Using protected method: %h", get_protected_key());

        // Cannot access local/private members
        // $display("Secret: %h", secret_code); // Error!
    endfunction
endclass
```

Object Lifetime

```
class Resource;
    static int instance_count = 0;
    int id;
    string name;

    function new(string name);
        instance_count++;
        id = instance_count;
        this.name = name;
        $display("Resource %0d (%s) created", id, name);
    endfunction

    // Destructor-like method (called explicitly)
    function void cleanup();
        $display("Resource %0d (%s) cleaned up", id, name);
        // Custom cleanup code here
    endfunction

    static function int get_instance_count();
        return instance_count;
    endfunction
endclass

module test_lifetime;
    Resource res1, res2;

    initial begin
        $display("Initial count: %0d", Resource::get_instance_count());

        res1 = new("Resource1");
        res2 = new("Resource2");

        $display("After creation: %0d", Resource::get_instance_count());

        // Objects are automatically garbage collected when no longer referenced
        res1 = null; // Remove reference
```

```
        // Explicit cleanup (if needed)
        res2.cleanup();
        res2 = null;

        // Note: instance_count doesn't decrease (no automatic destructor)
        $display("Final count: %0d", Resource::get_instance_count());
    end
endmodule
```

Static Members

Static members belong to the class rather than to individual instances. They are shared among all objects of the class.

Static Properties and Methods

```
class IDGenerator;
    static int next_id = 1;
    static int total_objects = 0;
    static string class_version = "v1.0";

    int object_id;
    string name;

    function new(string name);
        this.name = name;
        this.object_id = next_id++;
        total_objects++;
        $display("Created object %0d: %s", object_id, name);
    endfunction

    // Static method - can be called without creating an object
    static function int get_next_id();
        return next_id;
    endfunction

    static function int get_total_objects();
        return total_objects;
    endfunction

    static function void reset_counter();
        next_id = 1;
        total_objects = 0;
        $display("ID counter reset");
    endfunction

    // Static method to get class information
    static function string get_class_info();
        return $sformatf("IDGenerator %s - Next ID: %0d, Total: %0d",
                         class_version, next_id, total_objects);
    endfunction

    // Instance method that uses static data
```

```
    function void show_info();
        $display("Object %0d (%s) - Class has %0d total objects",
                 object_id, name, total_objects);
    endfunction
endclass

// Usage of static members
module test_static;
    IDGenerator obj1, obj2, obj3;

    initial begin
        // Call static method without creating objects
        $display("Class info: %s", IDGenerator::get_class_info());
        $display("Next ID will be: %0d", IDGenerator::get_next_id());

        // Create objects
        obj1 = new("First");
        obj2 = new("Second");
        obj3 = new("Third");

        // Show object info
        obj1.show_info();
        obj2.show_info();

        // Access static members through class name
        $display("Total objects created: %0d", IDGenerator::get_total_objects());

        // Reset static data
        IDGenerator::reset_counter();

        // Create new object after reset
        obj1 = new("After Reset");
        $display("Final class info: %s", IDGenerator::get_class_info());
    end
endmodule
```

Static vs Instance Members

```
class BankAccount;
    static real interest_rate = 0.05;  // Static - same for all accounts
    static int total_accounts = 0;     // Static - count of all accounts

    int account_number;                // Instance - unique per account
    real balance;                      // Instance - individual balance
    string owner_name;                 // Instance - individual owner

    function new(string name, real initial_balance = 0.0);
        total_accounts++;
        account_number = total_accounts;
        owner_name = name;
        balance = initial_balance;
    endfunction

    // Static method to change interest rate for all accounts
```

```
    static function void set_interest_rate(real new_rate);
        interest_rate = new_rate;
        $display("Interest rate changed to %.2f%% for all accounts",
                 new_rate * 100);
    endfunction

    // Instance method that uses both static and instance data
    function void apply_interest();
        real interest = balance * interest_rate;
        balance += interest;
        $display("Account %0d (%s): Interest $%.2f applied, new balance $%.2f",
                 account_number, owner_name, interest, balance);
    endfunction

    // Instance method
    function void deposit(real amount);
        balance += amount;
        $display("Account %0d: Deposited $%.2f, balance now $%.2f",
                 account_number, amount, balance);
    endfunction

    static function void print_statistics();
        $display("Bank Statistics:");
        $display("  Total accounts: %0d", total_accounts);
        $display("  Current interest rate: %.2f%%", interest_rate * 100);
    endfunction
endclass

module test_static_vs_instance;
    BankAccount acc1, acc2, acc3;

    initial begin
        // Create accounts
        acc1 = new("Alice", 1000.0);
        acc2 = new("Bob", 500.0);
        acc3 = new("Charlie", 1500.0);

        // Show initial statistics
        BankAccount::print_statistics();

        // Apply interest with current rate
        acc1.apply_interest();
        acc2.apply_interest();
        acc3.apply_interest();

        // Change interest rate (affects all accounts)
        BankAccount::set_interest_rate(0.08);

        // Apply new interest rate
        acc1.apply_interest();
        acc2.apply_interest();
        acc3.apply_interest();

        // Final statistics
        BankAccount::print_statistics();
```

```
    end
endmodule
```

Best Practices

1. **Use constructors** to initialize object state properly
2. **Implement deep copy methods** when objects contain dynamic arrays or other objects
3. **Use `this` keyword** to resolve naming conflicts and improve code clarity
4. **Apply appropriate access control** (public, protected, local) to encapsulate data
5. **Use static members** for class-wide data and utility functions
6. **Implement cleanup methods** for resources that need explicit cleanup
7. **Design classes with single responsibility** for better maintainability

Summary

Classes and objects in SystemVerilog provide powerful abstraction mechanisms for creating reusable and maintainable code. Key concepts include:

- **Class declarations** define templates for objects with properties and methods
- **Object creation** uses constructors and the `new()` operator
- **The `this` keyword** provides explicit reference to the current object
- **Class scope** controls member visibility and access
- **Static members** are shared across all instances of a class
- **Proper encapsulation** and access control improve code reliability

Understanding these concepts is essential for effective object-oriented programming in SystemVerilog, particularly for complex verification environments and testbenches.

SystemVerilog Classes and Objects - Simple Examples

Based on Chapter 9: Classes and Objects, here are super simple examples organized by subchapter:

Class Declarations

Basic Class Template

Simple class structure showing the fundamental syntax with properties and a basic method.

Simple Packet Class

Basic class representing a network packet with header, payload, and display functionality.

Transaction with Constructor

Class demonstrating constructor usage with default parameters and basic validation.

Properties and Methods

Data Packet Properties

Class showcasing different types of properties: basic types, arrays, random variables with constraints.

Network Packet Methods

Demonstrates different method types: constructor, function, task, virtual, and static methods.

Method Return Types

Examples of methods that return values vs. methods that perform actions without returning data.

Object Creation and Destruction

Configuration Block Objects

Simple example of creating multiple objects with different constructor parameters.

Object Assignment Demo

Shows the difference between shallow copy (handle assignment) and deep copy methods.

Buffer Management

Demonstrates object creation, modification, and memory management concepts.

The this Keyword

Counter Class with this

Class using `this` keyword to resolve naming conflicts between parameters and properties.

Object Self-Reference

Examples of methods that return references to the current object using `this`.

Object Comparison

Method that compares the current object with another object using `this`.

Class Scope and Lifetime

Access Control Demo

Class with public, protected, and local (private) members showing visibility rules.

Secure Data Class

Demonstrates encapsulation with different access levels and methods to access private data.

Resource Tracking

Class that tracks object creation and cleanup with instance counting.

Static Members

ID Generator Class

Class using static properties to generate unique IDs and track total object count.

Static Utility Methods

Examples of static methods that can be called without creating class instances.

Bank Account System

Demonstrates the difference between static (shared) and instance (individual) data members.

Best Practices Examples

Constructor Best Practices

Proper initialization of object state with parameter validation and default values.

Deep Copy Implementation

How to implement proper deep copy methods for objects containing dynamic arrays.

Encapsulation Example

Well-designed class showing proper use of access control and getter/setter methods.

Resource Management

Class design patterns for objects that need explicit cleanup or resource management.

Single Responsibility

Example of a class designed with a single, well-defined purpose for better maintainability.

Static vs Instance Usage

Clear examples showing when to use static members vs instance members appropriately.