Chapter 10: Inheritance and Polymorphism

Introduction to Object-Oriented Programming in SystemVerilog

SystemVerilog supports object-oriented programming (OOP) concepts that enable code reusability, modularity, and better organization. This chapter explores inheritance and polymorphism, two fundamental OOP concepts that allow you to create hierarchical relationships between classes and write more flexible, maintainable code.

Class Inheritance (extends)

Inheritance allows you to create new classes based on existing classes, inheriting their properties and methods while adding new functionality or modifying existing behavior.

Basic Inheritance Syntax

```
class BaseClass;
    // Base class properties and methods
endclass

class DerivedClass extends BaseClass;
    // Derived class inherits from BaseClass
    // Additional properties and methods
endclass
```

Practical Example: Vehicle Hierarchy

```
// Base Vehicle class
class Vehicle;
    string make;
    string model;
    int year;

    function new(string mk = "Unknown", string md = "Unknown", int yr = 2024);
        make = mk;
        model = md;
        year = yr;
    endfunction

    virtual function void display_info();
        $display("Vehicle: %s %s (%0d)", make, model, year);
    endfunction

    virtual function void start_engine();
        $display("Starting engine...");
    endfunction
```

```
endclass

// Car class inherits from Vehicle
class Car extends Vehicle;
    int num_doors;
    string fuel_type;

    function new(string mk = "Unknown", string md = "Unknown",
                 int yr = 2024, int doors = 4, string fuel = "Gasoline");
        super.new(mk, md, yr);  // Call parent constructor
        num_doors = doors;
        fuel_type = fuel;
    endfunction

    // Override parent method
    virtual function void display_info();
        super.display_info();  // Call parent method
        $display("  Type: Car, Doors: %0d, Fuel: %s", num_doors, fuel_type);
    endfunction

    function void open_trunk();
        $display("Opening car trunk...");
    endfunction
endclass

// Motorcycle class inherits from Vehicle
class Motorcycle extends Vehicle;
    bit has_sidecar;
    int engine_cc;

    function new(string mk = "Unknown", string md = "Unknown",
                 int yr = 2024, bit sidecar = 0, int cc = 250);
        super.new(mk, md, yr);
        has_sidecar = sidecar;
        engine_cc = cc;
    endfunction

    virtual function void display_info();
        super.display_info();
        $display("  Type: Motorcycle, Engine: %0dcc, Sidecar: %s",
                 engine_cc, has_sidecar ? "Yes" : "No");
    endfunction

    virtual function void start_engine();
        $display("Kick-starting motorcycle engine...");
    endfunction
endclass
```

Method Overriding

Method overriding allows derived classes to provide specific implementations of methods defined in their parent classes.

Rules for Method Overriding

1. The method signature must match exactly
2. Use the `virtual` keyword in the base class
3. The derived class method automatically becomes virtual

Example: Shape Hierarchy with Method Overriding

```
// Base Shape class
class Shape;
    string name;

    function new(string n = "Shape");
        name = n;
    endfunction

    // Virtual method to be overridden
    virtual function real calculate_area();
        $display("Warning: calculate_area() not implemented for %s", name);
        return 0.0;
    endfunction

    virtual function void draw();
        $display("Drawing a generic %s", name);
    endfunction
endclass

// Rectangle class
class Rectangle extends Shape;
    real width, height;

    function new(real w = 1.0, real h = 1.0);
        super.new("Rectangle");
        width = w;
        height = h;
    endfunction

    // Override calculate_area method
    virtual function real calculate_area();
        return width * height;
    endfunction

    virtual function void draw();
        $display("Drawing rectangle: %0.2f x %0.2f", width, height);
    endfunction
endclass

// Circle class
class Circle extends Shape;
    real radius;

    function new(real r = 1.0);
        super.new("Circle");
        radius = r;
    endfunction
```

```
    virtual function real calculate_area();
        return 3.14159 * radius * radius;
    endfunction

    virtual function void draw();
        $display("Drawing circle with radius: %0.2f", radius);
    endfunction
endclass

// Triangle class
class Triangle extends Shape;
    real base, height;

    function new(real b = 1.0, real h = 1.0);
        super.new("Triangle");
        base = b;
        height = h;
    endfunction

    virtual function real calculate_area();
        return 0.5 * base * height;
    endfunction

    virtual function void draw();
        $display("Drawing triangle: base=%0.2f, height=%0.2f", base, height);
    endfunction
endclass
```

The super Keyword

The super keyword provides access to the parent class's methods and properties from within a derived class.

Uses of super

1. **Calling parent constructor**: `super.new()`
2. **Calling parent methods**: `super.method_name()`
3. **Accessing parent properties**: `super.property_name`

Example: Employee Hierarchy

```
class Employee;
    string name;
    int employee_id;
    real base_salary;

    function new(string n, int id, real salary);
        name = n;
        employee_id = id;
        base_salary = salary;
    endfunction

    virtual function real calculate_pay();
```

```
        return base_salary;
    endfunction

    virtual function void display_info();
        $display("Employee: %s (ID: %0d), Base Salary: $%0.2f",
                 name, employee_id, base_salary);
    endfunction
endclass

class Manager extends Employee;
    real bonus_percentage;
    int team_size;

    function new(string n, int id, real salary, real bonus = 0.15, int team = 5);
        super.new(n, id, salary);  // Call parent constructor
        bonus_percentage = bonus;
        team_size = team;
    endfunction

    virtual function real calculate_pay();
        real base_pay = super.calculate_pay();  // Get base salary from parent
        return base_pay + (base_pay * bonus_percentage);
    endfunction

    virtual function void display_info();
        super.display_info();  // Call parent display method
        $display("  Role: Manager, Team Size: %0d, Bonus: %0.1f%%",
                 team_size, bonus_percentage * 100);
    endfunction

    function void conduct_meeting();
        $display("%s is conducting a team meeting", name);
    endfunction
endclass

class Developer extends Employee;
    string programming_language;
    int projects_completed;

    function new(string n, int id, real salary, string lang = "SystemVerilog");
        super.new(n, id, salary);
        programming_language = lang;
        projects_completed = 0;
    endfunction

    virtual function real calculate_pay();
        real base_pay = super.calculate_pay();
        real project_bonus = projects_completed * 500.0;  // $500 per project
        return base_pay + project_bonus;
    endfunction

    virtual function void display_info();
        super.display_info();
        $display("  Role: Developer, Language: %s, Projects: %0d",
                 programming_language, projects_completed);
```

```
    endfunction

    function void complete_project();
        projects_completed++;
        $display("%s completed a project in %s", name, programming_language);
    endfunction
endclass
```

Virtual Methods

Virtual methods enable polymorphism by allowing method calls to be resolved at runtime based on the actual object type.

Virtual Method Rules

1. Use `virtual` keyword in the base class method declaration
2. Derived class methods that override virtual methods are automatically virtual
3. Virtual methods enable dynamic binding

Example: Communication Protocol Stack

```
// Base Protocol class
class Protocol;
    string protocol_name;
    int header_size;

    function new(string name = "Generic", int hdr_size = 0);
        protocol_name = name;
        header_size = hdr_size;
    endfunction

    // Virtual methods for protocol operations
    virtual function void encode_packet(ref bit [7:0] data[]);
        $display("Generic encoding for %s protocol", protocol_name);
    endfunction

    virtual function void decode_packet(ref bit [7:0] data[]);
        $display("Generic decoding for %s protocol", protocol_name);
    endfunction

    virtual function int get_overhead();
        return header_size;
    endfunction

    virtual function void display_info();
        $display("Protocol: %s, Header Size: %0d bytes", protocol_name, header_size);
    endfunction
endclass

// TCP Protocol
class TCP_Protocol extends Protocol;
    int sequence_number;
    int window_size;
```

```
    function new();
        super.new("TCP", 20);  // TCP header is 20 bytes minimum
        sequence_number = 0;
        window_size = 65535;
    endfunction

    virtual function void encode_packet(ref bit [7:0] data[]);
        $display("TCP: Adding sequence number %0d and checksum", sequence_number);
        sequence_number++;
    endfunction

    virtual function void decode_packet(ref bit [7:0] data[]);
        $display("TCP: Verifying checksum and sequence number");
    endfunction

    virtual function int get_overhead();
        return super.get_overhead() + 4;  // Additional TCP options
    endfunction
endclass

// UDP Protocol
class UDP_Protocol extends Protocol;
    function new();
        super.new("UDP", 8);  // UDP header is 8 bytes
    endfunction

    virtual function void encode_packet(ref bit [7:0] data[]);
        $display("UDP: Adding simple header with length and checksum");
    endfunction

    virtual function void decode_packet(ref bit [7:0] data[]);
        $display("UDP: Basic header validation");
    endfunction
endclass

// HTTP Protocol (application layer)
class HTTP_Protocol extends Protocol;
    string method;
    string url;

    function new(string http_method = "GET", string request_url = "/");
        super.new("HTTP", 0);  // Variable header size
        method = http_method;
        url = request_url;
    endfunction

    virtual function void encode_packet(ref bit [7:0] data[]);
        $display("HTTP: Creating %s request for %s", method, url);
    endfunction

    virtual function void decode_packet(ref bit [7:0] data[]);
        $display("HTTP: Parsing request/response headers");
    endfunction

    virtual function int get_overhead();
```

```
        return method.len() + url.len() + 20;  // Estimated header size
    endfunction
endclass
```

Abstract Classes

While SystemVerilog doesn't have explicit abstract class syntax, you can create abstract-like behavior using pure virtual methods and base classes that shouldn't be instantiated directly.

Abstract Class Pattern

```
// Abstract Database Connection class
class DatabaseConnection;
    string connection_string;
    bit connected;

    function new(string conn_str);
        connection_string = conn_str;
        connected = 0;
    endfunction

    // Pure virtual methods (must be implemented by derived classes)
    pure virtual function bit connect();
    pure virtual function void disconnect();
    pure virtual function string execute_query(string query);
    pure virtual function bit is_connected();

    // Concrete method that can be inherited
    function void log_operation(string operation);
        $display("[%0t] Database Operation: %s", $time, operation);
    endfunction
endclass

// MySQL Database implementation
class MySQLConnection extends DatabaseConnection;
    int port;
    string database_name;

    function new(string host, int p = 3306, string db = "test");
        super.new($sformatf("mysql://%s:%0d/%s", host, p, db));
        port = p;
        database_name = db;
    endfunction

    // Implement abstract methods
    virtual function bit connect();
        log_operation("Connecting to MySQL");
        connected = 1;
        $display("Connected to MySQL database: %s", database_name);
        return connected;
    endfunction

    virtual function void disconnect();
        if (connected) begin
```

```
            log_operation("Disconnecting from MySQL");
            connected = 0;
            $display("Disconnected from MySQL");
        end
    endfunction

    virtual function string execute_query(string query);
        if (!connected) begin
            $display("Error: Not connected to database");
            return "";
        end
        log_operation($sformatf("Executing: %s", query));
        return "MySQL query result";
    endfunction

    virtual function bit is_connected();
        return connected;
    endfunction
endclass

// PostgreSQL Database implementation
class PostgreSQLConnection extends DatabaseConnection;
    string schema_name;

    function new(string host, string schema = "public");
        super.new($sformatf("postgresql://%s/%s", host, schema));
        schema_name = schema;
    endfunction

    virtual function bit connect();
        log_operation("Connecting to PostgreSQL");
        connected = 1;
        $display("Connected to PostgreSQL schema: %s", schema_name);
        return connected;
    endfunction

    virtual function void disconnect();
        if (connected) begin
            log_operation("Disconnecting from PostgreSQL");
            connected = 0;
            $display("Disconnected from PostgreSQL");
        end
    endfunction

    virtual function string execute_query(string query);
        if (!connected) begin
            $display("Error: Not connected to database");
            return "";
        end
        log_operation($sformatf("Executing on schema %s: %s", schema_name, query));
        return "PostgreSQL query result";
    endfunction

    virtual function bit is_connected();
        return connected;
```

```
    endfunction
endclass
```

Polymorphism Examples

Polymorphism allows objects of different types to be treated uniformly through a common interface, with method calls resolved at runtime based on the actual object type.

Example 1: Graphics Rendering System

```
// Test module demonstrating polymorphism with shapes
module polymorphism_demo;

    // Array of shape handles (polymorphic collection)
    Shape shapes[];
    Rectangle rect;
    Circle circ;
    Triangle tri;

    initial begin
        // Create different shape objects
        rect = new(5.0, 3.0);
        circ = new(2.5);
        tri = new(4.0, 6.0);

        // Store them in polymorphic array
        shapes = new[3];
        shapes[0] = rect;  // Rectangle assigned to Shape handle
        shapes[1] = circ;  // Circle assigned to Shape handle
        shapes[2] = tri;   // Triangle assigned to Shape handle

        $display("=== Polymorphic Shape Processing ===");

        // Process all shapes polymorphically
        foreach (shapes[i]) begin
            $display("\nShape %0d:", i+1);
            shapes[i].draw();  // Calls appropriate draw method
            $display("Area: %0.2f", shapes[i].calculate_area());  // Calls appropriate calcul
        end

        // Calculate total area
        real total_area = 0.0;
        foreach (shapes[i]) begin
            total_area += shapes[i].calculate_area();
        end
        $display("\nTotal area of all shapes: %0.2f", total_area);
    end
endmodule
```

Example 2: Network Protocol Handler

```
// Protocol handler demonstrating polymorphism
class NetworkStack;
    Protocol protocols[];

    function new();
        protocols = new[3];
        protocols[0] = new TCP_Protocol();
        protocols[1] = new UDP_Protocol();
        protocols[2] = new HTTP_Protocol("POST", "/api/data");
    endfunction

    // Process packet through all protocol layers
    function void process_packet(ref bit [7:0] data[]);
        $display("=== Processing Packet Through Network Stack ===");

        foreach (protocols[i]) begin
            $display("\n--- Layer %0d ---", i+1);
            protocols[i].display_info();
            protocols[i].encode_packet(data);
            $display("Overhead: %0d bytes", protocols[i].get_overhead());
        end

        $display("\n=== Decoding Packet ===");
        // Decode in reverse order
        for (int i = protocols.size()-1; i >= 0; i--) begin
            $display("\n--- Layer %0d Decode ---", i+1);
            protocols[i].decode_packet(data);
        end
    end

    function int calculate_total_overhead();
        int total = 0;
        foreach (protocols[i]) begin
            total += protocols[i].get_overhead();
        end
        return total;
    endfunction
endclass

// Test module for network protocols
module network_demo;
    NetworkStack stack;
    bit [7:0] packet_data[];

    initial begin
        stack = new();
        packet_data = new[100];  // 100-byte data packet

        // Initialize packet with dummy data
        foreach (packet_data[i]) begin
            packet_data[i] = i % 256;
        end
```

```
        // Process packet through protocol stack
        stack.process_packet(packet_data);

        $display("\nTotal Protocol Overhead: %0d bytes",
                 stack.calculate_total_overhead());
    end
endmodule
```

Example 3: Employee Management System

```
// Payroll system demonstrating polymorphism
class PayrollSystem;
    Employee employees[];

    function new();
        Manager mgr;
        Developer dev1, dev2;

        // Create different types of employees
        mgr = new("Alice Johnson", 1001, 75000.0, 0.20, 8);
        dev1 = new("Bob Smith", 1002, 65000.0, "SystemVerilog");
        dev2 = new("Carol Davis", 1003, 68000.0, "Python");

        // Add some completed projects for developers
        dev1.complete_project();
        dev1.complete_project();
        dev2.complete_project();
        dev2.complete_project();
        dev2.complete_project();

        // Store in polymorphic array
        employees = new[3];
        employees[0] = mgr;
        employees[1] = dev1;
        employees[2] = dev2;
    endfunction

    function void process_payroll();
        real total_payroll = 0.0;

        $display("=== Monthly Payroll Processing ===\n");

        foreach (employees[i]) begin
            real pay = employees[i].calculate_pay();  // Polymorphic call
            total_payroll += pay;

            employees[i].display_info();  // Polymorphic call
            $display("Monthly Pay: $%0.2f\n", pay);
        end

        $display("Total Monthly Payroll: $%0.2f", total_payroll);
    endfunction

    function void display_employee_details();
```

```
        $display("=== Employee Details ===\n");

        foreach (employees[i]) begin
            employees[i].display_info();

            // Type checking and casting for specific methods
            if ($cast(mgr_handle, employees[i])) begin
                Manager mgr_handle;
                mgr_handle.conduct_meeting();
            end else if ($cast(dev_handle, employees[i])) begin
                Developer dev_handle;
                dev_handle.complete_project();
            end
            $display("");
        end
    endfunction
endclass

// Test module for payroll system
module payroll_demo;
    PayrollSystem payroll;

    initial begin
        payroll = new();
        payroll.process_payroll();
        $display("\n" + {50{"="}});
        payroll.display_employee_details();
    end
endmodule
```

Best Practices and Design Patterns

Liskov Substitution Principle

Objects of derived classes should be substitutable for objects of the base class without altering program correctness.

```
// Good: Circle can substitute Shape
Shape my_shape = new Circle(5.0);
real area = my_shape.calculate_area();  // Works correctly
```

Interface Segregation

Keep interfaces focused and cohesive.

```
// Instead of one large interface, use specific interfaces
class Drawable;
    pure virtual function void draw();
endclass

class Calculable;
    pure virtual function real calculate_area();
endclass
```

```
class Circle extends Drawable, Calculable;  // Multiple inheritance
    // Implementation
endclass
```

Factory Pattern Example

```
class ShapeFactory;
    static function Shape create_shape(string shape_type, real param1 = 1.0, real param2 = 1.
        case (shape_type.tolower())
            "circle": return new Circle(param1);
            "rectangle": return new Rectangle(param1, param2);
            "triangle": return new Triangle(param1, param2);
            default: begin
                $display("Unknown shape type: %s", shape_type);
                return null;
            end
        endcase
    endfunction
endclass

// Usage
Shape my_shape = ShapeFactory::create_shape("circle", 3.0);
```

Common Pitfalls and Debugging Tips

Forgetting virtual keyword

```
// Wrong: Method won't be overridden properly
function void my_method();  // Not virtual

// Correct: Use virtual for overrideable methods
virtual function void my_method();
```

Incorrect super usage

```
// Wrong: Calling super incorrectly
function new();
    super();  // Syntax error

// Correct: Proper super call
function new();
    super.new();  // Correct syntax
```

Handle assignment vs object copying

```
// This copies the handle, not the object
Shape shape1 = new Circle(5.0);
Shape shape2 = shape1;  // Both handles point to same object
```

Summary

This chapter covered the essential object-oriented programming concepts in SystemVerilog:

- **Inheritance**: Creating new classes based on existing ones using `extends`
- **Method Overriding**: Providing specific implementations in derived classes
- **super keyword**: Accessing parent class methods and properties
- **Virtual Methods**: Enabling runtime method resolution for polymorphism
- **Abstract Classes**: Creating base classes with pure virtual methods
- **Polymorphism**: Treating objects of different types uniformly through common interfaces

These concepts enable you to write more modular, maintainable, and extensible SystemVerilog code by leveraging the power of object-oriented design principles. Understanding inheritance and polymorphism is crucial for building complex verification environments and reusable code libraries.

SystemVerilog OOP - Simple Examples by Topic

Class Inheritance (extends)

1. Basic Pet Hierarchy

Description: Simple Animal base class with Dog and Cat derived classes. Shows basic inheritance syntax with name and age properties.

2. Electronic Device Family

Description: Device base class with Phone and Laptop derived classes. Demonstrates inheriting common properties like brand, model, and power_on() method.

3. Simple Food Chain

Description: Food base class with Fruit and Vegetable derived classes. Shows inheritance of basic properties like name, color, and nutritional_value.

Method Overriding

4. Sound Maker Animals

Description: Animal base class with make_sound() method. Dog, Cat, and Cow classes override to make "Woof!", "Meow!", and "Moo!" respectively.

5. Transportation Methods

Description: Transport base class with move() method. Car, Bike, and Plane classes override to show "Driving...", "Pedaling...", and "Flying...".

6. Simple Math Operations

Description: Calculator base class with calculate() method. Add, Subtract, and Multiply classes override to perform specific operations.

The super Keyword

7. Student Grade System

Description: Student base class with name and basic info. GraduateStudent extends it, using super.new() to call parent constructor and super methods for grade calculation.

8. Bank Account Hierarchy

Description: Account base class with balance operations. SavingsAccount uses super to call parent methods while adding interest calculation.

9. Simple Game Characters

Description: Character base class with health and name. Warrior class uses super to initialize base stats then adds specific warrior abilities.

Virtual Methods

10. Simple Drawing Shapes

Description: Shape base class with virtual draw() method. Circle, Square, and Triangle override to show different ASCII art representations.

11. Basic File Handlers

Description: FileHandler base class with virtual process() method. TextFile, ImageFile, and VideoFile classes provide specific processing logic.

12. Simple Game Units

Description: Unit base class with virtual attack() method. Archer, Knight, and Wizard override to show different attack patterns and damage.

Abstract Classes (Pure Virtual)

13. Basic Database Operations

Description: Database base class with pure virtual connect(), query(), and disconnect() methods. MySQL and SQLite classes implement these methods.

14. Simple Printer Interface

Description: Printer base class with pure virtual print() method. LaserPrinter and InkjetPrinter implement specific printing mechanisms.

15. Basic Sensor Interface

Description: Sensor base class with pure virtual read_value() method. TemperatureSensor and HumiditySensor implement specific reading logic.

Polymorphism Examples

16. Pet Care System

Description: Array of Animal handles containing different pet types. Demonstrates polymorphic method calls for feeding and care routines.

17. Simple Media Player

Description: MediaFile array with different file types (Audio, Video, Image). Shows polymorphic play() method calls for different media types.

18. Basic Shape Calculator

Description: Collection of Shape objects calculating total area. Demonstrates polymorphic area calculation for mixed shape types.

19. Simple Command Processor

Description: Command base class with execute() method. Different command types (Save, Load, Delete) stored in array and executed polymorphically.

20. Basic Vehicle Fleet

Description: Vehicle array with cars, trucks, and motorcycles. Shows polymorphic fuel calculation and maintenance scheduling.

Design Patterns

21. Simple Factory Pattern

Description: AnimalFactory creates different animals based on string input. Demonstrates object creation without exposing instantiation logic.

22. Basic Observer Pattern

Description: Simple WeatherStation notifies multiple Display objects. Shows basic publisher-subscriber relationship using inheritance.

23. Simple Strategy Pattern

Description: SortStrategy base class with different sorting algorithms. Context class uses different strategies polymorphically.

Common Pitfalls Examples

24. Virtual Keyword Demo

Description: Side-by-side comparison showing difference between virtual and non-virtual methods in inheritance hierarchy.

25. Handle vs Object Copying

Description: Simple example demonstrating the difference between copying object handles and actual object duplication.

26. Proper super Usage

Description: Shows correct and incorrect ways to call parent constructors and methods using the super keyword.