SYSTEM VERILOG

INTERVIEW QUESTIONS

PART 2

JAIRAJ MIRASHI
DESIGN VERIFICATION ENGINEER

1. What is OOPS?

OOPS stands for Object-Oriented Programming System. It is a programming paradigm that focuses on organizing code into objects, which are instances of classes. OOPS emphasizes concepts such as encapsulation, inheritance, and polymorphism to structure and design software systems.

- ✓ For more detailed explanation refer this post - LINK

2. What is inheritance?

SYSTEMVERILOG INHERITANCE

- ❖ Inheritance is a concept in object-oriented programming (OOP) that allows us to extend a class to create another class and have access to all the properties and methods of the original parent class from the handle of the child class object.
- ❖ A derived class may add new properties and methods or modify the inherited properties and methods. Inheritance promotes reusability, as the derived class includes the properties and methods of the parent class, ready to use.
- ❖ If a class is derived from a derived class, it is referred to as multilevel inheritance.

PARENT CLASS:

1) It's an existing class.
2) It is the class whose features are inherited.
3) The parent class is also known as a base class or superclass.

CHILD CLASS:

1) It's an extended class.
2) It is the class that inherits the other class.
3) The child class is also known as an extended class, derived class, or subclass.

Example:

```
// Parent class definition
class parent_class;
    bit [31:0] addr;  // Property to store address
endclass

// Child class extending the parent class
class child_class extends parent_class;
    bit [31:0] data;  // Additional property to store data
endclass
// Module to demonstrate inheritance
module inheritance;
    initial begin
        child_class c = new();  // Creating an object of child class
        c.addr = 10;  // Accessing and assigning value to the inherited
property
```

```
        c.data = 20;  // Assigning value to the property in the child class
        $display("Value of addr = %0d, data = %0d", c.addr, c.data);  //
Displaying the values
    end
endmodule
```

3. How to write a message to a string in SystemVerilog?

```
module MessageExample;

  initial begin
    string message;
    int value = 42;
    string name = "John Doe";

    // Write a message to the string
    message = $sformatf("The value is %0d and the name is %s", value, name);

    // Display the message
    $display("Message: %s", message);
  end
endmodule
```

4. Signals inside the interface should be wires or logic?

Signals inside the interface in SystemVerilog can be declared as either wire or logic data types.

- In SystemVerilog, the most preferred data type for signals inside an interface is logic.
- The logic data type is modern and versatile, providing better support for modeling constructs like procedural assignments, arithmetic operations, and bit-level manipulations.
- It is recommended for new designs and when utilizing advanced SystemVerilog features such as assertions, coverage, and functional coverage.
- However, the wire data type is still commonly used, particularly for backward compatibility with older Verilog designs and when interfacing with external modules or IP blocks.
- The choice between wire and logic depends on specific design requirements and project/organization guidelines.
- Always refer to design specifications and guidelines to select the appropriate data type for interface signals.

5. Give examples of a static cast and a dynamic cast.

Static Casting:

Static casting is a concept in SystemVerilog that allows the conversion of one data type to another compatible data type. It is a compile-time operation, where the data type is fixed during compilation. Static casting is used to explicitly change the type of a value, variable, or expression.

Example of Static Casting:

In the following example, we demonstrate static casting by converting a real type into an int type. We multiply two real numbers, which results in a real value. Then, we convert the result into an int type using static casting and assign it to a variable of int type.

```
module casting;
    real r_a;
    int i_a;

    initial begin
        r_a = (2.1 * 3.2);

        // Real to integer conversion using static casting
        i_a = int'(2.1 * 3.2); // or i_a = int'(r_a);

        $display("Real value is %f", r_a);
        $display("Int value is %d", i_a);
    end
endmodule
```

Dynamic casting

Dynamic casting is a concept in SystemVerilog used to safely cast a handle of a parent class to a handle of a child class in a class hierarchy. It allows runtime checking of the compatibility of objects during casting and can result in a runtime error if the casting is not compatible.

Dynamic casting is performed using the $cast method, which checks the compatibility of the casting during runtime rather than compile-time.

Let's explore how dynamic casting works:

- It is always legal to assign a child class handle to a handle of a parent class higher in the inheritance tree.

For example:

```
parent_class = child_class; // Allowed
```

- It is never legal to directly assign a parent class handle to a handle of one of its child classes.

For example:

- `child_class = parent_class; // Not allowed`
- However, it is legal to assign a parent class handle to a child class handle if the parent class handle refers to an object of the child class.

For example:

```
parent_class p_h;
child_class c_h1, c_h2;
p_h = c_h1;
c_h2 = p_h; // Allowed because parent_class is pointing to child_class.
```

- Although assigning a parent class handle to a child class handle is allowed when the parent class handle refers to the child class, a compilation error is encountered due to incompatible types for the assignment. This can be overcome by using the $cast method.

For example:

```
$cast(c_h2, p_h); // Allowed using $cast method
```

Dynamic casting allows for safe casting between related classes in a class hierarchy, providing flexibility in handling class objects and enabling polymorphic behavior.

6. What is Mailbox?

A mailbox is a communication mechanism that facilitates the exchange of messages between processes in SystemVerilog. It allows data to be sent from one process and retrieved by another process. Here is an example that demonstrates how to declare and create a mailbox:

```
mailbox mbxRcv;
mbxRcv = new();
```

In SystemVerilog, there are two methods available for placing a message in a mailbox:

put() (blocking): This method blocks the executing process until there is space available in the mailbox to hold the message. Once space becomes available, the message is placed in the mailbox.

peek() (nonblocking): This method attempts to place the message in the mailbox but does not block the executing process. If there is space available, the message is successfully placed; otherwise, it returns without modifying the contents of the mailbox.

Similarly, to retrieve a message from a mailbox, there are two methods available:

get() (blocking): This method blocks the executing process until a message is available in the mailbox. Once a message becomes available, it is retrieved from the mailbox.

try_get() (nonblocking): This method attempts to retrieve a message from the mailbox but does not block the executing process. If a message is available, it is successfully retrieved; otherwise, it returns without modifying the contents of the receiving variable.

Additionally, the mailbox provides the num() method, which allows you to retrieve the number of messages currently present in the mailbox.

The usage of mailboxes and their associated methods enables synchronized and efficient communication between processes in SystemVerilog. It allows for the exchange of data in a controlled manner, ensuring seamless coordination between different parts of the design.

The following example demonstrates the usage of a mailbox for communication between a generator and a driver in a SystemVerilog design.

```
// Packet


class packet;
  rand bit [7:0] addr;
  rand bit [7:0] data;

  // Displaying randomized values
  function void post_randomize();
    $display("Packet::Packet Generated");
    $display("Packet::Addr=%0d, Data=%0d", addr, data);
  endfunction
endclass

// Generator - Generates the transaction packet and sends it to the driver

class generator;
  packet pkt;
  mailbox m_box;

  // Constructor, getting mailbox handle
  function new(mailbox m_box);
    this.m_box = m_box;
  endfunction

  task run;
    repeat(2) begin
      pkt = new();
      pkt.randomize(); // Generating packet
      m_box.put(pkt);  // Putting packet into the mailbox
      $display("Generator::Packet Put into Mailbox");
      #5;
    end
  endtask
endclass


// Driver - Retrieves the packet from the generator and displays its fields


class driver;
  packet pkt;
  mailbox m_box;

  // Constructor, getting mailbox handle
  function new(mailbox m_box);
    this.m_box = m_box;
  endfunction

  task run;
    repeat(2) begin
      m_box.get(pkt); // Retrieving packet from the mailbox
      $display("Driver::Packet Received");
      $display("Driver::Addr=%0d, Data=%0d\n", pkt.addr, pkt.data);
    end
  endtask
endclass


// Top-level module


module mailbox_ex;
  generator gen;
  driver dri;
  mailbox m_box; // Declaring mailbox m_box

  initial begin
    // Creating the mailbox and passing the same handle to the generator and
driver,
    // as the same mailbox should be shared for communication.
    m_box = new(); // Creating mailbox
    gen = new(m_box); // Creating generator and passing mailbox handle
    dri = new(m_box); // Creating driver and passing mailbox handle
    $display("------------------------------------------");
    fork
      gen.run(); // Process-1
      dri.run(); // Process-2
    join
    $display("------------------------------------------");
  end
endmodule
```

7. What will be the values of rand and randc variables if randomization fails?

If randomization fails for a rand variable, the value of the variable will depend on its initial value or any value assigned to it before randomization.

For example:

```
rand int myRandVar = 5; // Initial value assigned

// Randomization process
if (!myRandVar.randomize()) begin
  $display("Randomization failed for myRandVar");
  // Value of myRandVar will still be 5
end
```

In this example, if the randomization process for myRandVar fails, the value of myRandVar will remain as 5 since it was explicitly assigned that value before the randomization attempt.

On the other hand, for a randc variable, if randomization fails, it will be assigned an illegal value that cannot be legally represented by its data type.

Here's an example:

```
randc logic [3:0] myRandcVar;
// Randomization process
if (!myRandcVar.randomize()) begin
  $display("Randomization failed for myRandcVar");
  // Value of myRandcVar will be an illegal value
}
```

In this example, if the randomization process for myRandcVar fails, the variable will be assigned an illegal value that cannot be represented by the logic [3:0] data type. The specific illegal value assigned will depend on the simulation tool and implementation.

It's important to handle randomization failures appropriately in your code and check for success/failure using constructs like the randomize method or the randomize with construct. This allows you to perform error handling, provide fallback mechanisms, or implement alternative strategies when randomization fails.

8. Explain about the timeunit, timeprecision, and timescale in SystemVerilog.

The timeunit, timeprecision, and timescale directives in SystemVerilog are used to specify the time resolution and units for simulation.

The timeunit directive sets the basic time unit for simulation. It determines the smallest time increment used in the simulation. The default timeunit is 1ns.

For example:

```
`timescale 1ns/1ns
```

The timeprecision directive determines the number of decimal places used for fractional time values. The default timeprecision is 1ns.

For example:

```
`timescale 1ns/100ps
```

The timescale directive combines both timeunit and timeprecision directives in a single statement. It sets the time unit and time precision for the entire compilation unit.

For example:

```
`timescale 1ns/1ps
```

By specifying the timescale directive at the beginning of a file, you can define the time unit and precision used throughout the design.

9. How to randomize a real data type variable?

In SystemVerilog, to randomize a real data type variable, you can make use of the built-in randomize method provided by the language. The randomize method is used to assign random values to variables based on their data type constraints.

To randomize a real variable, you need to follow these steps:

1) Declare a real variable that you want to randomize.
2) Call the randomize method on the real variable to assign a random value.

Here's an example code snippet demonstrating how to randomize a real variable in SystemVerilog:

```
class MyClass;
  real myRealVar; // Declare a real variable

  function new();
    myRealVar = $random; // Assign a random value to myRealVar using $random
system function
  endfunction

  function void displayValue();
    $display("Randomized value of myRealVar: %f", myRealVar); // Display the
randomized value
  endfunction
endclass

module Example;
  initial begin
    MyClass obj = new(); // Create an instance of MyClass
    obj.displayValue(); // Call the displayValue function to print the
randomized value
  end
endmodule
```

10. What is the default value of an enumerated data type?

The default value of an enumerated data type is the first value defined in the enumeration list. If no explicit assignment is given, the first enumeration member is assigned a value of 0, and subsequent members are assigned values in the order they are defined, incrementing by 1.

For example, consider the following enumeration:

```
typedef enum logic [2:0] {RED, GREEN, BLUE} Color;
```

In this case, the default value of the Color data type would be RED, which has an implicit value of 0.

If you explicitly assign values to the enumeration members, the default value remains the same, unless you specifically set it otherwise.

11. What is polymorphism?

Polymorphism

Polymorphism, which means "many forms," is a concept in SystemVerilog that allows an object to take on multiple forms. In SystemVerilog, polymorphism enables a variable of the parent class type to hold objects of child classes and access the methods of those child classes directly from the parent class variable. It also allows a child class method to have a different definition than its parent class if the parent class method is declared as virtual.

Here are the key points regarding polymorphism in SystemVerilog:

1) A parent class variable can be used to reference objects of child classes.
2) The parent class method should be declared as virtual.
3) The method name should be the same in both the parent and child classes.
4) Child class objects can be assigned to a parent class variable using the assignment operator.
5) The child class methods can be accessed using the parent class handle.

12. Give an example of polymorphism.

Let's see an example that demonstrates polymorphism in SystemVerilog:

```
// Base class
class base_class;
    virtual function void display();
        $display("Inside base class");
    endfunction
endclass
```

```
// Child class 1
class child_class_1 extends base_class;
    function void display();
        $display("Inside child class 1");
    endfunction
endclass

// Child class 2
class child_class_2 extends base_class;
    function void display();
        $display("Inside child class 2");
    endfunction
endclass

module class_polymorphism;
    initial begin
        base_class b_h;                       // Declare base class handle
        child_class_1 c_1 = new();            // Create child class objects
        child_class_2 c_2 = new();

        b_h = c_1;                            // Assign child class to base
class handle
        b_h = c_2;

        b_h.display();                        // Access child class methods
using base class handle
    end
endmodule
```

13. What is meant by abstraction?

An abstract class in SystemVerilog is declared using the keyword "virtual" and is intended to serve as a prototype for its derived subclasses. Here are some key points about abstract classes:

An abstract class sets out the prototype for its sub-classes, defining common attributes and behaviors that the derived classes should implement.

An abstract class cannot be directly instantiated; it can only be derived to create objects of its derived classes.

An abstract class may contain methods for which only the prototype is provided without any implementation. These methods are referred to as abstract methods or pure virtual methods, and they serve as placeholders for the derived classes to provide their own implementation.

Let's look at examples illustrating the concepts of instantiating and deriving from an abstract class:

Instantiating a Virtual Class:

In this example, we attempt to create an object of a virtual class, which results in a compilation error because abstract classes cannot be instantiated.

```
// Abstract class
virtual class packet;
  bit [31:0] addr;
endclass

class packet_2 extends packet;
  // Derived class implementation
endclass

module virtual_class;
  initial begin
    packet_2 p;
    p = new(); // Compilation error: Abstract class cannot be instantiated
    p.addr = 12;
  end
endmodule
```

```
Error: Instantiation of the object 'p' cannot be done because its type
'packet' is an abstract base class.
Perhaps there is a derived class that should be used.
```

Deriving from a Virtual Class:

In this example, we derive a class from an abstract class, providing an implementation for the abstract method defined in the abstract class.

```
/ Abstract class
virtual class packet;
  bit [31:0] addr;
endclass

class extended_packet extends packet;
  function void display;
    $display("Value of addr is %0d", addr);
  endfunction
endclass

module virtual_class;
  initial begin
    extended_packet p;
    p = new();
    p.addr = 10;
    p.display(); // Calling the derived class method
  end
endmodule
```

```
Value of addr is 10
```

Abstraction is an important concept in SystemVerilog and helps in designing modular and extensible code by defining common behavior in abstract classes while allowing specific implementations in derived classes.

14. What are virtual methods?

Virtual Methods

Virtual methods in SystemVerilog are declared using the keyword "virtual" before the method declaration. There are two types of virtual methods:

1) Virtual functions
2) Virtual tasks.

Here's an overview of each:

- Virtual Functions:

A function declared with the virtual keyword before the function keyword is referred to as a virtual function. Virtual functions allow polymorphic behavior, where the derived class can override the implementation of the function defined in the base class.

Virtual function syntax:

```
/virtual function return_type function_name;
  // Function definition
endfunction
```

- Virtual Tasks:

A task declared with the virtual keyword before the task keyword is referred to as a virtual task. Similar to virtual functions, virtual tasks also support polymorphism, allowing the derived class to provide its own implementation of the task defined in the base class.

Virtual task syntax:

```
virtual task task_name;
  // Task definition
endtask
```

About Virtual Methods:

When a virtual method is used, a base class handle can be assigned an object of a derived class. The behavior of calling the virtual method depends on whether the method is virtual in the base class or not.

Consider the following scenario:

```
base_class     b_c;
extended_class e_c;
// Assigning the derived class object to the base class handle
b_c = e_c;

// Calling the display() method
b_c.display();
```

- ✓ If the display() method in the base class is declared as virtual, then the display() method of the extended class will be called.
- ✓ If the display() method in the base class is not declared as virtual, then the display() method of the base class will be called.

Let's look at examples illustrating the usage of virtual methods:

- ➢ Method without the Virtual Keyword:

In this example, the method inside the base class is declared without the virtual keyword. When calling the method of the base class using the base class handle pointing to the extended class, the base class method will be invoked.

```
// Parent class
class base_class;
  function void display;
    $display("Inside base_class");
  endfunction
endclass

// Child class
class extended_class extends base_class;
  function void display;
    $display("Inside extended_class");
  endfunction
endclass

module virtual_class;
  initial begin
    base_class     b_c;
    extended_class e_c;

    e_c = new();
    b_c = e_c;

    b_c.display();
  end
endmodule
```

```
Simulator Output:
Inside base_class
```

- Method with the Virtual Keyword:

In this example, the method inside the base class is declared with the virtual keyword. When calling the method of the base class using the base class handle pointing to the extended class, the extended class method will be invoked.

```
// Parent class
class base_class;
  virtual function void display;
    $display("Inside base_class");
  endfunction
endclass

// Child class
class extended_class extends base_class;
  function void display;
    $display("Inside extended_class");
  endfunction
endclass

module virtual_class;
  initial begin
    base_class     b_c;
    extended_class e_c;

    e_c = new();
    b_c = e_c;

    b_c.display();
  end
endmodule
```

```
Simulator Output:
Inside extended_class
```

In this example, the display() method is declared as virtual in the base class, allowing the derived class method to be called when using the base class handle.

15. What is a virtual class?

- **Refer to Question - 13**

16. What is a scope resolution operator?

The scope-resolution operator (::) in SystemVerilog is used to access static members or members of an outer scope within a class or module. It allows you to explicitly specify the scope of the member you want to access.

Regarding static class members, the scope-resolution operator is used to access static properties or methods of a class without creating an object of that class. Here's how the scope-resolution operator is used:

```
<class_name>::<member_name>
```

By using the scope-resolution operator, you can directly reference the static members of a class using the class name followed by the operator (::) and the member name.

For example, in the given code snippet:

```
class A;
  static int i;
endclass

program main;
  A obj_1;

  initial begin
    obj_1.i = 123;
    $display(A::i);
  end
endprogram
```

The line obj_1.i = 123; assigns the value 123 to the static property i of class A using an instance of A. However, to display the value of i, we can directly access it using the scope-resolution operator with the class name: $display(A::i);. This way, we can access the static member i without creating an object of class A.

The scope-resolution operator is particularly useful when you want to access static members or members of an outer scope within a class or module, allowing you to explicitly specify the scope of the member you want to access. It helps in resolving naming conflicts and accessing members that are not specific to a particular instance but are associated with the class itself.

17. What is shallow copy?

Shallow Copy refers to the process of copying the properties of a main class object without copying the object of its subclass. In a shallow copy, the copied object shares the same reference to the subclass object as the original object.

Let's consider the following example:

```
class sub;
    int addr;
endclass

class main;
    int data;
    sub sub_h = new();
endclass

module tb;
    main M1, M2;
    initial
        begin
            M1 = new;                      // Create a new instance of the main
class M1
            M1.data = 4;                   // Set the value of data property in
M1 to 4
            M1.sub_h.addr = 5;             // Set the value of addr property in
M1's sub_h object to 5

            M2 = new M1;                   // Perform a shallow copy by
assigning M1 to M2
        end
endmodule
```

In the above code, we have a main class and a sub class. The main class contains an integer variable "data" and an instance of the sub class "sub_h". The sub class has an integer variable "addr". The tb module creates two instances of the main class, M1 and M2.

During the initialization, M1's data is set to 4, and the addr variable of M1's sub_h object is set to 5. Then, M2 is assigned the value of M1.

In a shallow copy, only the properties of the main class are copied, not the object of the subclass. So, after the assignment M2 = new M1;, M2 will have its own copy of the main class properties, including the "data" variable. However, M2 will still share the same reference to the sub_h object as M1. This means that any changes made to the sub_h object in M2 will also affect the sub_h object in M1.

It's important to note that shallow copy is the default behavior when assigning objects in SystemVerilog. To perform a deep copy, where both the main class properties and the subclass objects are copied, additional code or techniques, such as a custom copy constructor or assignment operator, would need to be implemented.

18. What is deep copy?

Deep copying in SystemVerilog involves copying all the properties of the main class as well as its subclass. Unlike shallow copy, which only copies nested class handles, deep copy replicates the entire object hierarchy. To achieve deep copying, a custom method needs to be implemented.

In a deep copy, a new object is created, and all the properties of the class are copied to the new handle. This ensures that both the main class and its nested class members are duplicated. By performing a full or deep copy, any modifications made to the copied object will not affect the original object.

Here is an example of a custom deep copy method in SystemVerilog:

```
class Sub;
    int addr;  // Property to store the address

    // Custom copy method to perform deep copy
    function Sub copy;
        Sub copy_obj = new;  // Create a new instance of Sub class
        copy_obj.addr = this.addr;  // Copy the address value
        return copy_obj;  // Return the copied object
    endfunction
endclass

class Main;
    int data;  // Property to store the data
    Sub sub_h = new;  // Instance of the Sub class

    // Custom copy method to perform deep copy
    function Main copy;
        Main copy_obj = new;  // Create a new instance of Main class
        copy_obj.data = this.data;  // Copy the data value
        copy_obj.sub_h = sub_h.copy;  // Perform deep copy of sub_h using custom
copy method
        return copy_obj;  // Return the copied object
    endfunction
endclass

module tb;
    Main M1, M2;  // Objects of the Main class

    initial
        begin
            M1 = new;  // Instantiate M1 object
            M1.data = 4;  // Set the data value of M1
            M1.sub_h.addr = 5;  // Set the addr value of M1's sub_h object

            M2 = M1.copy;  // Perform deep copy by assigning M1 to M2
            M2.sub_h.addr = 10;  // Modify the addr value of M2's sub_h object
        end
endmodule
```

In this code, we define two classes: Sub and Main. The Sub class has an integer property addr to store the address. It also includes a custom copy method to perform a deep copy. Inside the copy method, a new instance of Sub called copy_obj is created, and the addr value is copied from the current object using this.addr. Finally, the copied object copy_obj is returned.

The Main class contains an integer property data and an instance of the Sub class called sub_h, which represents a nested object. It also includes a custom copy method to perform a deep copy. Inside the copy method, a new instance of Main called copy_obj is created, and the data value is copied from the current object using this.data. The sub_h object is deep copied by invoking its custom copy method and assigning the copied object to copy_obj.sub_h. Finally, the copied object copy_obj is returned.

In the tb module, we instantiate two objects: M1 and M2 of the Main class. Initially, we create M1 using the new keyword, set the data value to 4, and modify the addr value of the sub_h object. Then, we perform a deep copy by assigning M1 to M2. After the copy, we modify the addr value of the sub_h object in M2.

By implementing the deep copy mechanism, each object maintains its independent copies of class properties, including nested objects. This ensures that modifications made to one object do not affect the other object, providing isolation and preventing unintended side effects.

19. What is method overriding?

- ❖ Overriding class members allows child classes to redefine or replace properties and methods inherited from the parent class.
- ❖ Child classes can provide their own implementation of class members while still inheriting the structure and functionality from the parent class.
- ❖ Class members with the same name as those in the parent class are declared in the child class to override them.
- ❖ When an object of the child class is created, accessing the overridden member will invoke the implementation defined in the child class.
- ❖ Overriding class members provides flexibility and customization in the behavior of child classes, allowing them to modify or enhance the inherited functionality to suit their specific requirements.
- ❖ It enables the creation of specialized classes based on a common parent class.

```
// Overriding class members
// Base class or parent class properties and methods can be overridden in the
child class or extended class.
// Defining the class properties and methods with the same name as the parent
class in the child class will override the class members.

// Overriding class member example

// Parent class
class parent_class;
  bit [31:0] addr;

  // Parent class display() method
  function display();
    $display("Addr = %0d", addr);
  endfunction
endclass

// Child class extending the parent class
class child_class extends parent_class;
  bit [31:0] data;

  // Child class display() method overriding the parent class method
  function display();
    $display("Data = %0d", data);
  endfunction
endclass

module inheritance;
  initial begin
    child_class c = new();
    c.addr = 10;
    c.data = 20;
    c.display();
  end
endmodule
```

In the provided example, we have a parent class called parent_class and a child class called child_class that extends the parent class. The parent class has a property addr of type bit [31:0] and a display() method that displays the value of addr.

In the child class, we define a property data of type bit [31:0]. We also redefine the display() method in the child class to display the value of data. This is an example of method overriding, where the child class provides its own implementation of the display() method, which overrides the implementation in the parent class.

In the inheritance module, we create an instance of the child class c using the new() keyword. We assign a value of 10 to the addr property of c and a value of 20 to the data property. Finally, we call the display() method on c, which invokes the overridden method in the child class.

The simulator output will display:

Data = 20

20. What is method overloading?

Method overloading in SystemVerilog refers to the ability to have multiple methods with the same name in a class, but with different parameter lists. Each overloaded method performs a different operation based on the specific parameters provided.

Let's extend the previous code example to demonstrate method overloading:

```
class ParentClass;
  bit [31:0] addr;

  function void displayAddr();
    $display("Addr = %0d", addr);
  endfunction

  function void displayValue(int value);
    $display("Value = %0d", value);
  endfunction
endclass

class ChildClass extends ParentClass;
```

```
  bit [31:0] data;

  function void displayData();
    $display("Data = %0d", data);
  endfunction

  function void displayDataValue(int value);
    $display("Data = %0d, Value = %0d", data, value);
  endfunction
endclass

module Inheritance;
  initial begin
    ChildClass c = new();
    c.addr = 10;
    c.data = 20;
    c.displayData();                // Calls the displayData() method in
ChildClass
    c.displayDataValue(30);         // Calls the displayDataValue(int value)
method in ChildClass
    c.displayAddr();                // Calls the displayAddr() method in
ParentClass
  end
endmodule
```

The ParentClass defines two functions: displayAddr() and displayValue(). displayAddr() simply displays the value of the addr property, while displayValue() takes an integer argument and displays its value.

The ChildClass extends ParentClass and adds two additional functions: displayData() and displayDataValue(). These functions operate on the data property of the ChildClass and provide their own display messages.

In the Inheritance module, an object of ChildClass named c is created.

The addr and data properties of c are assigned values (10 and 20 respectively).

c.displayData() is called, which invokes the displayData() method defined in ChildClass. It displays the message "Data = 20".

c.displayDataValue(30) is called, which invokes the displayDataValue(int value) method defined in ChildClass. It displays the message "Data = 20, Value = 30".

c.displayAddr() is called, which invokes the displayAddr() method defined in ParentClass. It displays the message "Addr = 10".

Override versus Overload

Override	Overload
Method override refers to the ability of an extended class to replace a base class method with a new implementation.	Method overload allows for the existence of multiple methods with the same name but different parameter lists.
In SystemVerilog, an extended class can override a base class method by defining a method with the same name in the extended class.	Overloading a method means providing multiple versions of the method that can accept different argument types or different numbers of arguments.
When an object of the extended class invokes the overridden method, the implementation in the extended class is executed instead of the base class method.	The appropriate version of the method is selected based on the arguments passed to the method.
The override completely replaces the base class method in the extended class.	However, in SystemVerilog, method and function overloading is not supported.
	Due to the complexity of working with loose data types in Verilog and SystemVerilog, overloading methods is not implemented in the language.

In summary, while method override is possible in SystemVerilog, method overload is not supported. Overriding a method replaces the base class method in the extended class, whereas overloading would allow for multiple versions of a method with different parameter lists, selected based on the arguments passed to the method. However, due to implementation challenges, method and function overloading is not available in SystemVerilog.