Virtual Prototyping Question Bank 1st Internals

Disclaimer:

This answer script contains solutions they may not cover all possible cases, edge conditions, or optimizations. Students are encouraged to understand the solutions thoroughly, modify them as needed, and apply them responsibly in their studies. The correctness, completeness, and effectiveness of the solutions are subject to verification by students and educators. The authors and providers of this answer script do not assume any liability for the consequences, direct or indirect, arising from the use of these solutions.

---

1. ***Write a C++ program to find the maximum number in an array of Integers.***

```
#include <iostream>

int findMax(int arr[], int size) {
    int max = arr[0]; // Assume the first element is the maximum
    for (int i = 1; i < size; ++i) {
        if (arr[i] > max) {
            max = arr[i]; // Update max if a larger element is found
        }
    }
    return max;
}

int main() {
    int arr[] = {10, 5, 20, 15, 25};
    int size = sizeof(arr) / sizeof(arr[0]); // Calculate the size of the array
    int maximum = findMax(arr, size);
    std::cout << "The maximum number in the array is: " << maximum << std::endl;
    return 0;
}
```

---

2. ***Write a C++ program to find out the sum of digits of a number.***

```
#include <iostream>

int sumOfDigits(int number) {
      int sum = 0;
      while (number != 0) {
      sum += number % 10; // Add the last digit to the sum
      number /= 10; // Remove the last digit from the number
      }
      return sum;
```

```cpp
}

int main() {
      int num;
      std::cout << "Enter a number: ";
      std::cin >> num;
      int sum = sumOfDigits(num);
      std::cout << "The sum of digits of " << num << " is: " <<
sum << std::endl;
      return 0;
}
```

---

3. *Write C++ program to print the reverse of an array.*

```cpp
#include <iostream>

void reverseArray(int arr[], int size) {
      for (int i = size - 1; i >= 0; --i) {
      std::cout << arr[i] << " ";
      }
      std::cout << std::endl;
}

int main() {
      int arr[] = {1, 2, 3, 4, 5};
      int size = sizeof(arr) / sizeof(arr[0]);

      std::cout << "Original array: ";
      for (int i = 0; i < size; ++i) {
      std::cout << arr[i] << " ";
      }
      std::cout << std::endl;

      std::cout << "Reverse of the array: ";
      reverseArray(arr, size);

      return 0;
}
```

---

4. *Write a C++ code to swap values of two variables using pointers.*

```cpp
#include <iostream>

void swap(int* ptr1, int* ptr2) {
      int temp = *ptr1;
      *ptr1 = *ptr2;
      *ptr2 = temp;
}

int main() {
      int a = 5, b = 10;

      std::cout << "Before swapping:\n";
      std::cout << "a = " << a << ", b = " << b << std::endl;

      // Pointers to the variables
      int* ptr_a = &a;
      int* ptr_b = &b;

      // Swap the values using pointers
      swap(ptr_a, ptr_b);

      std::cout << "After swapping:\n";
      std::cout << "a = " << a << ", b = " << b << std::endl;

      return 0;
}
```

---

5. *Write C++ program to find length of string using pointers.*

```
#include <iostream>

int stringLength(const char* str) {
      int length = 0;
      while (*str != '\0') {
      length++;
      str++;
      }
      return length;
}

int main() {
      const char* str = "Hello, World!";
      int length = stringLength(str);
      std::cout << "Length of the string \"" << str << "\" is: "
<< length << std::endl;
      return 0;
}
```

6. *What are virtual functions? Write a C++ program to demonstrate polymorphism. ---1*

Virtual functions in C++ are functions in a base class that are declared using the `virtual` keyword and can be overridden by derived classes. When a virtual function is called through a base class pointer or reference to a derived class object, the function that is called is determined at runtime based on the actual type of the object.

```cpp
#include <iostream>

// Base class
class Shape {
public:
      // Virtual function
      virtual void draw() {
      std::cout << "Drawing a Shape\n";
      }
};

// Derived class Circle
class Circle : public Shape {
public:
      // Override the virtual function
      void draw() override {
      std::cout << "Drawing a Circle\n";
      }
};

// Derived class Rectangle
class Rectangle : public Shape {
public:
      // Override the virtual function
      void draw() override {
      std::cout << "Drawing a Rectangle\n";
      }
};

int main() {
      Shape* shapes[2]; // Array of base class pointers

      Circle circle;
      Rectangle rectangle;
```

```
        // Assign derived class objects to base class pointers
        shapes[0] = &circle;
        shapes[1] = &rectangle;

        // Call the virtual function using base class pointers
        for (int i = 0; i < 2; ++i) {
        shapes[i]->draw(); // This call will be resolved at runtime
        }

        return 0;
}
```

In this program:

- We have a base class `Shape` with a virtual function `draw()`.
- We have two derived classes `Circle` and `Rectangle`, each overriding the `draw()` function.
- In the `main()` function, we create an array of base class pointers `shapes[]`.
- We instantiate objects of `Circle` and `Rectangle` classes.
- We assign the addresses of these objects to the base class pointers in the `shapes[]` array.
- We call the `draw()` function through the base class pointers. The actual function called depends on the type of object being pointed to. This is an example of polymorphism where the appropriate function is resolved at runtime.

---

7. ***Describe dynamic binding in C++ using an example.***

Dynamic binding, also known as late binding or runtime polymorphism, is a mechanism in C++ where the selection of the function to be called is determined at runtime rather than at compile time. This is achieved through the use of virtual functions and pointers or references to base class objects.

```
#include <iostream>

// Base class
class Animal {
public:
        // Virtual function
        virtual void sound() {
        std::cout << "Animal makes a sound\n";
        }
};

// Derived class Dog
```

```cpp
class Dog : public Animal {
public:
      // Override the virtual function
      void sound() override {
      std::cout << "Dog barks\n";
      }
};

// Derived class Cat
class Cat : public Animal {
public:
      // Override the virtual function
      void sound() override {
      std::cout << "Cat meows\n";
      }
};

int main() {
      Animal* animal; // Base class pointer

      Dog dog;
      Cat cat;

      animal = &dog;
      animal->sound(); // Calls Dog's sound() function dynamically

      animal = &cat;
      animal->sound(); // Calls Cat's sound() function dynamically

      return 0;
}
```

In this example:

- We have a base class `Animal` with a virtual function `sound()`.
- We have two derived classes `Dog` and `Cat`, each overriding the `sound()` function.
- In the `main()` function, we declare a base class pointer `animal`.
- We create objects of `Dog` and `Cat` classes.
- We assign the addresses of these objects to the base class pointer `animal`.
- We call the `sound()` function through the base class pointer. The actual function called depends on the type of object being pointed to. This is dynamic binding in action, where the appropriate function is resolved at runtime.

8. ***What is operator overloading. Overload + operator in the implementation of complex numbers. ----2***

Operator overloading in C++ allows you to redefine the behavior of operators when they are used with user-defined types. This means that you can make operators like `+`, `-`, `*`, etc., work with your own custom types, such as classes.

```
#include <iostream>

class Complex {
private:
      double real;
      double imag;

public:
      // Constructor
      Complex(double r = 0.0, double i = 0.0) : real(r), imag(i)
{}

      // Overload + operator
      Complex operator+(const Complex& other) const {
      return Complex(real + other.real, imag + other.imag);
      }

      // Function to display complex number
      void display() {
      std::cout << real << " + " << imag << "i";
      }
};

int main() {
      // Create two complex numbers
      Complex c1(2.5, 3.0);
      Complex c2(1.5, 2.0);

      // Add two complex numbers using overloaded operator
      Complex result = c1 + c2;

      // Display the result
      std::cout << "Result of addition: ";
      result.display();
      std::cout << std::endl;

      return 0;
}
```

In this implementation:

- We define a `Complex` class to represent complex numbers.
- Inside the class, we overload the `+` operator using the `operator+` member function. This function takes a constant reference to another complex number and returns a new complex number representing the sum.
- In the `main()` function, we create two `Complex` objects `c1` and `c2`, representing complex numbers.
- We use the overloaded `+` operator to add these two complex numbers, and store the result in another `Complex` object `result`.
- Finally, we display the result of addition using the `display()` member function.

---

9. *What is function overloading? Explain using an example in C++.*

Function overloading in C++ allows you to define multiple functions with the same name but with different parameter lists. This means that you can have multiple functions with the same name performing different tasks based on the number or type of parameters they accept.

```cpp
#include <iostream>

// Function to calculate the area of a square
double calculateArea(double side) {
      return side * side;
}

// Function to calculate the area of a rectangle
double calculateArea(double length, double width) {
      return length * width;
}

int main() {
      double side = 5.0;
      double length = 4.0;
      double width = 3.0;

      // Calculate the area of a square
      std::cout << "Area of square with side " << side << " is: "
<< calculateArea(side) << std::endl;

      // Calculate the area of a rectangle
      std::cout << "Area of rectangle with length " << length << "
and width " << width << " is: " << calculateArea(length, width) <<
std::endl;

      return 0;
}
```

In this example:

- We have two functions named `calculateArea`, but each takes a different set of parameters.
- The first `calculateArea` function takes a single parameter `side`, representing the side length of a square, and calculates the area of the square.
- The second `calculateArea` function takes two parameters `length` and `width`, representing the length and width of a rectangle, and calculates the area of the rectangle.
- In the `main()` function, we call these functions with different arguments to calculate the area of a square and a rectangle.
- Depending on the number and type of parameters passed to the `calculateArea` function, the appropriate overloaded version of the function is called. This is function overloading in action.

---

10. Differentiate between run time polymorphism and compile time polymorphism with an example in C++.

Runtime polymorphism and compile-time polymorphism are both mechanisms used in object-oriented programming to achieve polymorphism, but they operate at different stages of program execution and are implemented differently.

Compile-time polymorphism (Static polymorphism):

- Compile-time polymorphism refers to the polymorphism that is resolved during compile time.
- It is achieved through function overloading and operator overloading.
- The compiler determines which function to call based on the number and types of arguments provided at compile time.
- The decision about which function to execute is made at compile time and remains fixed during runtime.

```cpp
#include <iostream>

// Function to add two integers
int add(int a, int b) {
      return a + b;
}

// Function to add three integers
int add(int a, int b, int c) {
      return a + b + c;
}

int main() {
    std::cout << "Sum of 2 + 3 = " << add(2, 3) << std::endl; // Calls the first add function
    std::cout << "Sum of 2 + 3 + 4 = " << add(2, 3, 4) << std::endl; // Calls the second add function
    return 0;
}
```

In this example, the add function is overloaded to accept different numbers of parameters. The decision about which add function to call is made by the compiler based on the number of arguments provided.

Runtime polymorphism (Dynamic polymorphism):

- Runtime polymorphism refers to the polymorphism that is resolved during runtime.
- It is achieved through virtual functions and inheritance.
- The function to be executed is determined at runtime based on the type of object being referred to (not the type of pointer/reference).
- It allows for the implementation of function overriding in derived classes.

Example:

```
#include <iostream>

// Base class
class Animal {
public:
    // Virtual function
    virtual void sound() {
    std::cout << "Animal makes a sound\n";
    }
};

// Derived class Dog
class Dog : public Animal {
public:
    // Override the virtual function
    void sound() override {
    std::cout << "Dog barks\n";
    }
};

int main() {
    Animal* animal; // Base class pointer
    Dog dog; // Derived class object
```

```
      animal = &dog; // Pointing to the derived class object

      // Call the virtual function using the base class pointer
      animal->sound(); // Calls Dog's sound() function dynamically

      return 0;
}
```

In this example, `Animal` is the base class and `Dog` is the derived class. The `sound` function is declared as virtual in the base class. When `sound()` is called through a base class pointer pointing to a `Dog` object, the `sound()` function of `Dog` class is executed. This decision is made at runtime based on the type of object being referred to.

---

11. *What are constructors? Explain with an example for complex numbers using c++.*

Constructors in C++ are special member functions of a class that are automatically called when an object of that class is created. They are used to initialize the object's data members or perform any other initialization tasks.

```
#include <iostream>

class Complex {
private:
      double real;
      double imag;

public:
      // Default constructor
      Complex() {
      real = 0.0;
      imag = 0.0;
      }

      // Parameterized constructor
      Complex(double r, double i) {
      real = r;
      imag = i;
      }

      // Function to display complex number
      void display() {
      std::cout << real << " + " << imag << "i";
```

```
    }
};

int main() {
    // Create complex numbers using different constructors
    Complex c1; // Calls default constructor, initializes to
(0.0, 0.0)
    Complex c2(2.5, 3.0); // Calls parameterized constructor,
initializes to (2.5, 3.0)

    // Display the complex numbers
    std::cout << "c1 = ";
    c1.display();
    std::cout << std::endl;

    std::cout << "c2 = ";
    c2.display();
    std::cout << std::endl;

    return 0;
}
```

In this example:

- We have a class `Complex` representing complex numbers with two private data members `real` and `imag`.
- We define two constructors:
  - The default constructor initializes the complex number to (0.0, 0.0).
  - The parameterized constructor initializes the complex number with the provided real and imaginary parts.
- In the `main()` function, we create two complex numbers `c1` and `c2` using different constructors.
- We then display the complex numbers using the `display()` member function.

Constructors allow us to initialize objects of a class with initial values or perform any necessary setup during object creation.

---

12. Illustrate copy constructors with an example in c++. -----3

Copy constructors are special member functions in C++ that are used to initialize an object using another object of the same class. They are called when an object is initialized with another object of the same class, either by direct initialization or by passing the object by value to a function.

Here's an example illustrating copy constructors in C++:

```cpp
#include <iostream>

class MyClass {
private:
	int data;

public:
	// Default constructor
	MyClass() {
	std::cout << "Default constructor called\n";
	data = 0;
	}

	// Parameterized constructor
	MyClass(int d) {
	std::cout << "Parameterized constructor called\n";
	data = d;
	}

	// Copy constructor
	MyClass(const MyClass &obj) {
	std::cout << "Copy constructor called\n";
	data = obj.data;
	}

	// Function to display data
	void display() {
	std::cout << "Data: " << data << std::endl;
	}
};

// Function taking object by value
void function(MyClass obj) {
	std::cout << "Inside function\n";
	obj.display();
}

int main() {
	// Create objects using different constructors
	MyClass obj1; // Calls default constructor
	MyClass obj2(5); // Calls parameterized constructor

	// Create object using copy constructor
	MyClass obj3(obj2); // Calls copy constructor

	// Display data of objects
```

```
    obj1.display();
    obj2.display();
    obj3.display();

    // Pass object to function by value
    function(obj1);

    return 0;
}
```

In this example:

- We have a class `MyClass` with a default constructor, a parameterized constructor, and a copy constructor.
- The copy constructor is defined as `MyClass(const MyClass &obj)`, which takes a reference to another `MyClass` object as its parameter.
- In the `main()` function, we create objects `obj1` and `obj2` using the default and parameterized constructors respectively.
- We then create `obj3` by initializing it with `obj2`. This triggers the call to the copy constructor.
- We display the data of all three objects to verify the initialization.
- Finally, we pass `obj1` to a function `function()` by value. This also invokes the copy constructor as the object is passed by value.

```
Default constructor called
Parameterized constructor called
Copy constructor called
Data: 0
Data: 5
Data: 5
Inside function
Data: 0
```

As seen in the output, the copy constructor is called when objects are initialized with existing objects of the same class, either directly or through function arguments.

---

13. Illustrate function overriding with an example in C++.

Function overriding is a feature of object-oriented programming that allows a derived class to provide a specific implementation of a function that is already defined in its base class. When a derived class overrides a function, it provides its own implementation of that function which is used instead of the base class implementation when called through a base class pointer or reference.

```
#include <iostream>

// Base class
class Animal {
public:
      // Virtual function
      virtual void sound() {
      std::cout << "Animal makes a sound\n";
      }
};

// Derived class Dog
class Dog : public Animal {
public:
      // Override the virtual function
      void sound() override {
      std::cout << "Dog barks\n";
      }
};

// Derived class Cat
class Cat : public Animal {
public:
      // Override the virtual function
      void sound() override {
      std::cout << "Cat meows\n";
      }
};

int main() {
      // Base class pointer
      Animal* animal;

      Dog dog; // Derived class object
      Cat cat; // Derived class object

      animal = &dog; // Pointing to the derived class object
      animal->sound(); // Calls Dog's sound() function dynamically

      animal = &cat; // Pointing to the derived class object
      animal->sound(); // Calls Cat's sound() function dynamically

      return 0;
}
```

In this example:

- We have a base class `Animal` with a virtual function `sound()`.
- We have two derived classes `Dog` and `Cat`, each overriding the `sound()` function with their specific implementation.
- In the `main()` function, we declare a base class pointer `animal`.
- We create objects of `Dog` and `Cat` classes.
- We assign the addresses of these objects to the base class pointer `animal`.
- We call the `sound()` function through the base class pointer. The actual function called depends on the type of object being referred to. This is function overriding in action, where the appropriate function is resolved at runtime.

---

14. Overload << operator for date class using friend function. date class comprises of date, month and year. ----4

To overload the `<<` operator for a `Date` class, you can define a friend function that takes an `ostream` object and a `Date` object as parameters. Here's how you can implement it:

```
#include <iostream>

class Date {
private:
      int day;
      int month;
      int year;

public:
      Date(int d, int m, int y) : day(d), month(m), year(y) {}

      // Friend function to overload the << operator
      friend std::ostream& operator<<(std::ostream& os, const Date&
date);
};

// Overloading the << operator using a friend function
std::ostream& operator<<(std::ostream& os, const Date& date) {
      os << date.day << "/" << date.month << "/" << date.year;
      return os;
}

int main() {
      Date date(10, 2, 2024);
      std::cout << "Date: " << date << std::endl;

      return 0;
}
```

In this example:

- We define a Date class with private data members day, month, and year.
- We provide a constructor to initialize the Date object with day, month, and year values.
- We declare a friend function operator<< which takes an ostream object (std::ostream& os) and a Date object (const Date& date) as parameters.
- Inside the friend function, we use the os stream to output the day, month, and year of the Date object.
- In the main() function, we create a Date object and use the overloaded << operator to output the date to the console. The friend function is called implicitly due to its friendship with the Date class.

---

15. Illustrate the use of #define directive in C/C++.

The #define directive in C/C++ is a preprocessor directive used to define symbolic constants, macros, or functions. It allows you to give a name to a constant value or to define a short function-like structure.

Here are a few common uses of the #define directive:

Defining Constants: You can use #define to define constants. For example:

cpp

```
#define PI 3.14159
```

This will replace all occurrences of PI in your code with 3.14159.

Defining Macros: You can define macros using #define. For example:
cpp

```
#define SQUARE(x) ((x) * (x))
```

This defines a macro SQUARE which squares its argument. So, SQUARE(5) would be replaced with (5 * 5).

Defining Conditional Compilation Flags: You can use #define to define conditional compilation flags. For example:
cpp

```
#define DEBUG_MODE
```

You can then use `#ifdef` or `#ifndef` to check whether this flag is defined:

```
#ifdef DEBUG_MODE
    // Debugging code
#endif
```

1.

Here's a simple example demonstrating the use of `#define` to define a constant and a macro:

```
#include <iostream>

// Define a constant
#define PI 3.14159

// Define a macro to calculate the square of a number
#define SQUARE(x) ((x) * (x))

int main() {
    // Use the constant
    std::cout << "Value of PI: " << PI << std::endl;

    // Use the macro
    int num = 5;
    std::cout << "Square of " << num << " is: " << SQUARE(num) <<
std::endl;

    return 0;
}
```

In this example:

- We use `#define` to define a constant `PI` with the value `3.14159`.
- We use `#define` to define a macro `SQUARE` which squares its argument.
- In the `main()` function, we use both the constant `PI` and the macro `SQUARE` to perform calculations.

Design the following combinational circuits using systemc with monitor and driver

16. decoder 2:4 with enable

```
#include <systemc.h>

SC_MODULE(Decoder24) {
      sc_in<bool> enable;
      sc_in<sc_uint<2>> in;
      sc_out<sc_uint<4>> out;

      void decode() {
      if (enable.read()) {
            sc_uint<4> value = 0;
            value[in.read()] = 1;
            out.write(value);
      } else {
            out.write(0);
      }
      }

      SC_CTOR(Decoder24) {
      SC_METHOD(decode);
      sensitive << in << enable;
      }
};

// Monitor
SC_MODULE(MonitorDecoder24) {
      sc_in<sc_uint<4>> out;

      void monitor() {
      cout << "Decoder24 Output: " << out.read() << endl;
      }

      SC_CTOR(MonitorDecoder24) {
      SC_METHOD(monitor);
      sensitive << out;
      }
};

// Driver
SC_MODULE(DriverDecoder24) {
      sc_out<sc_uint<2>> in;
      sc_out<bool> enable;

      void drive() {
```

```
      wait(5, SC_NS);
      in.write(0);
      enable.write(true);
      wait(5, SC_NS);
      in.write(1);
      wait(5, SC_NS);
      in.write(2);
      wait(5, SC_NS);
      in.write(3);
      wait(5, SC_NS);
      enable.write(false);
      wait(5, SC_NS);
      in.write(2);
      wait(5, SC_NS);
      in.write(3);
      wait(5, SC_NS);
      enable.write(true);
      wait(5, SC_NS);
      in.write(1);
      wait(5, SC_NS);
      enable.write(false);
      wait(5, SC_NS);
      sc_stop();
      }

      SC_CTOR(DriverDecoder24) {
      SC_THREAD(drive);
      }
};

int sc_main(int argc, char* argv[]) {
      sc_signal<sc_uint<2>> in;
      sc_signal<bool> enable;
      sc_signal<sc_uint<4>> out;

      Decoder24 decoder("decoder24");
      decoder.in(in);
      decoder.enable(enable);
      decoder.out(out);

      MonitorDecoder24 monitor("monitor24");
      monitor.out(out);

      DriverDecoder24 driver("driver24");
      driver.in(in);
      driver.enable(enable);
```

```
        sc_start();
        return 0;
}
```

---

17. and gate

```
#include <systemc.h>

SC_MODULE(AndGate) {
        sc_in<bool> a;
        sc_in<bool> b;
        sc_out<bool> out;

        void andOperation() {
        out.write(a.read() && b.read());
        }

        SC_CTOR(AndGate) {
        SC_METHOD(andOperation);
        sensitive << a << b;
        }
};

// Monitor
SC_MODULE(MonitorAndGate) {
        sc_in<bool> out;

        void monitor() {
        cout << "AndGate Output: " << out.read() << endl;
        }

        SC_CTOR(MonitorAndGate) {
        SC_METHOD(monitor);
        sensitive << out;
        }
};

// Driver
SC_MODULE(DriverAndGate) {
        sc_out<bool> a, b;

        void drive() {
```

```
      wait(5, SC_NS);
      a.write(false);
      b.write(false);
      wait(5, SC_NS);
      a.write(true);
      wait(5, SC_NS);
      b.write(true);
      wait(5, SC_NS);
      a.write(false);
      wait(5, SC_NS);
      sc_stop();
      }

      SC_CTOR(DriverAndGate) {
      SC_THREAD(drive);
      }
};

int sc_main(int argc, char* argv[]) {
      sc_signal<bool> a, b, out;

      AndGate andGate("andGate");
      andGate.a(a);
      andGate.b(b);
      andGate.out(out);

      MonitorAndGate monitor("monitor");
      monitor.out(out);

      DriverAndGate driver("driver");
      driver.a(a);
      driver.b(b);

      sc_start();
      return 0;
}
```

---

18. nand gate

```
#include <systemc.h>

SC_MODULE(NandGate) {
      sc_in<bool> a;
      sc_in<bool> b;
      sc_out<bool> out;

      void nandOperation() {
      out.write(!(a.read() && b.read()));
      }

      SC_CTOR(NandGate) {
      SC_METHOD(nandOperation);
      sensitive << a << b;
      }
};

// Monitor
SC_MODULE(MonitorNandGate) {
      sc_in<bool> out;

      void monitor() {
      cout << "NandGate Output: " << out.read() << endl;
      }

      SC_CTOR(MonitorNandGate) {
      SC_METHOD(monitor);
      sensitive << out;
      }
};

// Driver
SC_MODULE(DriverNandGate) {
      sc_out<bool> a, b;

      void drive() {
      wait(5, SC_NS);
      a.write(false);
      b.write(false);
      wait(5, SC_NS);
      a.write(true);
      wait(5, SC_NS);
      b.write(true);
      wait(5, SC_NS);
      a.write(false);
      wait(5, SC_NS);
      sc_stop();
      }

      SC_CTOR(DriverNandGate) {
```

```
      SC_THREAD(drive);
      }
};

int sc_main(int argc, char* argv[]) {
      sc_signal<bool> a, b, out;

      NandGate nandGate("nandGate");
      nandGate.a(a);
      nandGate.b(b);
      nandGate.out(out);

      MonitorNandGate monitor("monitor");
      monitor.out(out);

      DriverNandGate driver("driver");
      driver.a(a);
      driver.b(b);

      sc_start();
      return 0;
}
```

---

19. half adder

```
#include <systemc.h>

SC_MODULE(HalfAdder) {
      sc_in<bool> a;
      sc_in<bool> b;
      sc_out<bool> sum;
      sc_out<bool> carry;

      void add() {
      sum.write(a.read() ^ b.read());
      carry.write(a.read() && b.read());
      }

      SC_CTOR(HalfAdder) {
      SC_METHOD(add);
      sensitive << a << b;
      }
};
```

```
// Monitor
SC_MODULE(MonitorHalfAdder) {
      sc_in<bool> sum, carry;

      void monitor() {
      cout << "HalfAdder Output: Sum = " << sum.read() << ", Carry = " << carry.read() << endl;
      }

      SC_CTOR(MonitorHalfAdder) {
      SC_METHOD(monitor);
      sensitive << sum << carry;
      }
};

// Driver
SC_MODULE(DriverHalfAdder) {
      sc_out<bool> a, b;

      void drive() {
      wait(5, SC_NS);
      a.write(false);
      b.write(false);
      wait(5, SC_NS);
      a.write(true);
      wait(5, SC_NS);
      b.write(true);
      wait(5, SC_NS);
      a.write(false);
      wait(5, SC_NS);
      sc_stop();
      }

      SC_CTOR(DriverHalfAdder) {
      SC_THREAD(drive);
      }
};

int sc_main(int argc, char* argv[]) {
      sc_signal<bool> a, b, sum, carry;

      HalfAdder halfAdder("halfAdder");
      halfAdder.a(a);
      halfAdder.b(b);
      halfAdder.sum(sum);
      halfAdder.carry(carry);

      MonitorHalfAdder monitor("monitor");
      monitor.sum(sum);
      monitor.carry(carry);

      DriverHalfAdder driver("driver");
      driver.a(a);
      driver.b(b);

      sc_start();
      return 0;
}
```

---

20. full adder

```
#include <systemc.h>

SC_MODULE(FullAdder) {
      sc_in<bool> a;
      sc_in<bool> b;
      sc_in<bool> cin;
      sc_out<bool> sum;
      sc_out<bool> carry;

      void add() {
      sum.write((a.read() ^ b.read()) ^ cin.read());
      carry.write((a.read() && b.read()) || (a.read() &&
cin.read()) || (b.read() && cin.read()));
      }

      SC_CTOR(FullAdder) {
      SC_METHOD(add);
      sensitive << a << b << cin;
      }
};

// Monitor
SC_MODULE(MonitorFullAdder) {
      sc_in<bool> sum, carry;

      void monitor() {
      cout << "FullAdder Output: Sum = " << sum.read() << ", Carry
= " << carry.read() << endl;
      }

      SC_CTOR(MonitorFullAdder) {
      SC_METHOD(monitor);
      sensitive << sum << carry;
      }
};

// Driver
SC_MODULE(DriverFullAdder) {
      sc_out<bool> a, b, cin;

      void drive() {
      wait(5, SC_NS);
      a.write(false);
      b.write(false);
      cin.write(false);
      wait(5, SC_NS);
      a.write(true);
      wait(5, SC_NS);
      b.write(true);
      wait(5, SC_NS);
      cin.write(true);
      wait(5, SC_NS);
      a.write(false);
      wait(5, SC_NS);
      sc_stop();
      }

      SC_CTOR(DriverFullAdder) {
      SC_THREAD(drive);
      }
};

int sc_main(int argc, char* argv[]) {
      sc_signal<bool> a, b, cin, sum, carry;

      FullAdder fullAdder("fullAdder");
      fullAdder.a(a);
      fullAdder.b(b);
      fullAdder.cin(cin);
      fullAdder.sum(sum);
      fullAdder.carry(carry);

      MonitorFullAdder monitor("monitor");
      monitor.sum(sum);
      monitor.carry(carry);

      DriverFullAdder driver("driver");
      driver.a(a);
      driver.b(b);
      driver.cin(cin);

      sc_start();
      return 0;
}
```

Design the following sequential circuit using systemc with monitor and driver

21. 4bit shft register

```
#include <systemc.h>

SC_MODULE(ShiftRegister) {
      sc_in<bool> clock;
      sc_in<bool> reset;
      sc_in<bool> input;
      sc_out<sc_uint<4>> output;

      sc_uint<4> registerValue;

      void shift() {
      if (reset.read()) {
            registerValue = 0;
      } else if (clock.posedge()) {
            registerValue = (registerValue << 1) | input.read();
      }
      output.write(registerValue);
      }

      SC_CTOR(ShiftRegister) {
      SC_METHOD(shift);
      sensitive << clock.pos() << reset << input;
      }
};

// Monitor
SC_MODULE(MonitorShiftRegister) {
      sc_in<sc_uint<4>> output;
      void monitor() {
      cout << "Shift Register Output: " << output.read() << endl;
      }

      SC_CTOR(MonitorShiftRegister) {
      SC_METHOD(monitor);
      sensitive << output;
      }
};

// Driver
SC_MODULE(DriverShiftRegister) {
      sc_out<bool> clock, reset;
      sc_out<bool> input;

      void drive() {
      wait(5, SC_NS);
      reset.write(true);
      wait(5, SC_NS);
      reset.write(false);
      wait(5, SC_NS);
      clock.write(true);
      wait(5, SC_NS);
      input.write(true);
      wait(5, SC_NS);
      input.write(false);
      wait(5, SC_NS);
      clock.write(false);
      wait(5, SC_NS);
      clock.write(true);
      wait(5, SC_NS);
      input.write(true);
      wait(5, SC_NS);
      input.write(false);
      wait(5, SC_NS);
      clock.write(false);
      wait(5, SC_NS);
      sc_stop();
      }

      SC_CTOR(DriverShiftRegister) {
      SC_THREAD(drive);
      }
};

int sc_main(int argc, char* argv[]) {
      sc_signal<bool> clock, reset, input;
      sc_signal<sc_uint<4>> output;

      ShiftRegister shiftRegister("shiftRegister");
      shiftRegister.clock(clock);
      shiftRegister.reset(reset);
      shiftRegister.input(input);
      shiftRegister.output(output);

      MonitorShiftRegister monitor("monitor");
      monitor.output(output);

      DriverShiftRegister driver("driver");
      driver.clock(clock);
      driver.reset(reset);
      driver.input(input);

      sc_start();
      return 0;
}
```

---

22. D flip flop

```
#include <systemc.h>

SC_MODULE(DFlipFlop) {
      sc_in<bool> d;
      sc_in<bool> clock;
      sc_out<bool> q;
      sc_out<bool> qBar;

      void flipFlop() {
      if (clock.posedge()) {
            q.write(d.read());
            qBar.write(!d.read());
      }
      }

      SC_CTOR(DFlipFlop) {
      SC_METHOD(flipFlop);
      sensitive << clock.pos();
      }
};
```

```
// Monitor
SC_MODULE(MonitorDFlipFlop) {
	sc_in<bool> q, qBar;

	void monitor() {
	cout << "D Flip Flop Output: Q = " << q.read() << ", QBar = " << qBar.read() << endl;
	}

	SC_CTOR(MonitorDFlipFlop) {
	SC_METHOD(monitor);
	sensitive << q << qBar;
	}
};

// Driver
SC_MODULE(DriverDFlipFlop) {
	sc_out<bool> d, clock;

	void drive() {
	wait(5, SC_NS);
	d.write(false);
	clock.write(false);
	wait(5, SC_NS);
	d.write(true);
	wait(5, SC_NS);
	clock.write(true);
	wait(5, SC_NS);
	clock.write(false);
	wait(5, SC_NS);
	d.write(false);
	wait(5, SC_NS);
	clock.write(true);
	wait(5, SC_NS);
	clock.write(false);
	wait(5, SC_NS);
	sc_stop();
	}

	SC_CTOR(DriverDFlipFlop) {
	SC_THREAD(drive);
	}
};

int sc_main(int argc, char* argv[]) {
	sc_signal<bool> d, clock, q, qBar;
```

```
      DFlipFlop flipFlop("flipFlop");
      flipFlop.d(d);
      flipFlop.clock(clock);
      flipFlop.q(q);
      flipFlop.qBar(qBar);

      MonitorDFlipFlop monitor("monitor");
      monitor.q(q);
      monitor.qBar(qBar);

      DriverDFlipFlop driver("driver");
      driver.d(d);
      driver.clock(clock);

      sc_start();
      return 0;
}
```

---

23. mod 8 counter

```
#include <systemc.h>

SC_MODULE(CounterMod8) {
      sc_in<bool> clock;
      sc_out<sc_uint<3>> count;

      sc_uint<3> counter;

      void countMod8() {
      if (clock.posedge()) {
            counter = (counter + 1) % 8;
            count.write(counter);
      }
      }

      SC_CTOR(CounterMod8) {
      SC_METHOD(countMod8);
      sensitive << clock.pos();
      }
};

// Monitor
SC_MODULE(MonitorCounterMod8) {
      sc_in<sc_uint<3>> count;

      void monitor() {
      cout << "Counter Mod 8 Output: " << count.read() << endl;
      }

      SC_CTOR(MonitorCounterMod8) {
      SC_METHOD(monitor);
      sensitive << count;
      }
};

// Driver
SC_MODULE(DriverCounterMod8) {
      sc_out<bool> clock;

      void drive() {
      for (int i = 0; i < 10; ++i) {
            wait(5, SC_NS);
            clock.write(false);
            wait(5, SC_NS);
            clock.write(true);
      }
      sc_stop();
      }

      SC_CTOR(DriverCounterMod8) {
      SC_THREAD(drive);
      }
};

int sc_main(int argc, char* argv[]) {
      sc_signal<bool> clock;
      sc_signal<sc_uint<3>> count;

      CounterMod8 counter("counter");
      counter.clock(clock);
      counter.count(count);

      MonitorCounterMod8 monitor("monitor");
      monitor.count(count);

      DriverCounterMod8 driver("driver");
      driver.clock(clock);

      sc_start();
      return 0;
}
```

24. t ff

```
#include <systemc.h>

SC_MODULE(TFlipFlop) {
      sc_in<bool> t;
      sc_in<bool> clock;
      sc_out<bool> q;
      sc_out<bool> qBar;

      bool currentState;

      void flipFlop() {
      if (clock.posedge()) {
            if (t.read() == 1) {
                  currentState = !currentState;
            }
            q.write(currentState);
            qBar.write(!currentState);
      }
      }

      SC_CTOR(TFlipFlop) {
      SC_METHOD(flipFlop);
      sensitive << clock.pos();
      }
};

// Monitor
SC_MODULE(MonitorTFlipFlop) {
      sc_in<bool> q, qBar;

      void monitor() {
      cout << "T Flip Flop Output: Q = " << q.read() << ", QBar = " << qBar.read() << endl;
      }

      SC_CTOR(MonitorTFlipFlop) {
      SC_METHOD(monitor);
      sensitive << q << qBar;
      }
```

```
};

// Driver
SC_MODULE(DriverTFlipFlop) {
      sc_out<bool> t, clock;

      void drive() {
      wait(5, SC_NS);
      t.write(false);
      clock.write(false);
      wait(5, SC_NS);
      t.write(true);
      wait(5, SC_NS);
      clock.write(true);
      wait(5, SC_NS);
      clock.write(false);
      wait(5, SC_NS);
      t.write(false);
      wait(5, SC_NS);
      clock.write(true);
      wait(5, SC_NS);
      clock.write(false);
      wait(5, SC_NS);
      sc_stop();
      }

      SC_CTOR(DriverTFlipFlop) {
      SC_THREAD(drive);
      }
};

int sc_main(int argc, char* argv[]) {
      sc_signal<bool> t, clock, q, qBar;

      TFlipFlop flipFlop("flipFlop");
      flipFlop.t(t);
      flipFlop.clock(clock);
      flipFlop.q(q);
      flipFlop.qBar(qBar);

      MonitorTFlipFlop monitor("monitor");
      monitor.q(q);
      monitor.qBar(qBar);

      DriverTFlipFlop driver("driver");
      driver.t(t);
```

```
        driver.clock(clock);

        sc_start();
        return 0;
}
```

---

25. sr flip flop

```
#include <systemc.h>

SC_MODULE(SRFlipFlop) {
        sc_in<bool> s;
        sc_in<bool> r;
        sc_in<bool> clock;
        sc_out<bool> q;
        sc_out<bool> qBar;

        bool currentState;

        void flipFlop() {
        if (clock.posedge()) {
                if (s.read() && !r.read()) {
                        currentState = true;
                } else if (!s.read() && r.read()) {
                        currentState = false;
                }
                q.write(currentState);
                qBar.write(!currentState);
        }
        }

        SC_CTOR(SRFlipFlop) {
        SC_METHOD(flipFlop);
        sensitive << clock.pos();
        }
};

// Monitor
SC_MODULE(MonitorSRFlipFlop) {
        sc_in<bool> q, qBar;

        void monitor() {
        cout << "SR Flip Flop Output: Q = " << q.read() << ", QBar =
" << qBar.read() << endl;
        }

        SC_CTOR(MonitorSRFlipFlop) {
        SC_METHOD(monitor);
        sensitive << q << qBar;
        }
};

// Driver
SC_MODULE(DriverSRFlipFlop) {
        sc_out<bool> s, r, clock;

        void drive() {
        wait(5, SC_NS);
        s.write(false);
        r.write(false);
        clock.write(false);
        wait(5, SC_NS);
        s.write(true);
        wait(5, SC_NS);
        clock.write(true);
        wait(5, SC_NS);
        clock.write(false);
        wait(5, SC_NS);
        r.write(true);
        wait(5, SC_NS);
        s.write(false);
        wait(5, SC_NS);
        clock.write(true);
        wait(5, SC_NS);
        clock.write(false);
        wait(5, SC_NS);
        sc_stop();
        }

        SC_CTOR(DriverSRFlipFlop) {
        SC_THREAD(drive);
        }
};

int sc_main(int argc, char* argv[]) {
        sc_signal<bool> s, r, clock, q, qBar;

        SRFlipFlop flipFlop("flipFlop");
        flipFlop.s(s);
      flipFlop.r(r);
      flipFlop.clock(clock);
      flipFlop.q(q);
      flipFlop.qBar(qBar);

      MonitorSRFlipFlop monitor("monitor");
      monitor.q(q);
      monitor.qBar(qBar);

      DriverSRFlipFlop driver("driver");
      driver.s(s);
      driver.r(r);
      driver.clock(clock);

      sc_start();
      return 0;
}
```
